Departamento de Informática
Centro Tecnológico
Universidade Federal de Minas Gerais

Curso de C

Sumário

1. INTRODUÇÃO

Vamos, neste curso, aprender os conceitos básicos da linguagem de programação C a qual tem se tornado cada dia mais popular, devido à sua versatilidade e ao seu poder. Uma das grandes vantagens do C é que ele possui tanto características de "alto nível" quanto de "baixo nível".
Apesar de ser bom, não é pré-requisito do curso um conhecimento anterior de linguagens de programação. É importante um familiaridade com computadores.
O C nasceu na década de 70. Seu inventor, Dennis Ritchie, implementou-o pela primeira vez usando um DEC PDP-11 rodando o sistema operacional UNIX. O C é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.
Veremos os comandos do ANSI C, o C padronizado pela ANSI. Além destes veremos algumas funções comuns em compiladores para diversos sistemas operacionais, com ênfase no sistema DOS. Quando não houver equivalentes para as funções em outros sistemas, apresentaremos formas alternativas de uso dos comandos.
Sugerimos que o aluno realmente use o máximo possível dos exemplos aqui apresentados, gerando os programas executáveis com o seu compilador. Quando utilizamos o compilador aprendemos a lidar com mensagens de aviso, mensagens de erro, bugs, etc. Apenas ler os exemplos não basta. O conhecimento de uma linguagem de programação transcede o conhecimento de estruturas e funções. O C exige, além do domínio da linguagem em si, uma familiaridade com o compilador e experiência em achar "bugs" nos programas. É importante então que o leitor digite, compile e execute os exemplos apresentados.

2. AULA 1 - Primeiros Passos

2.1. O C é "Case Sensitive"

Vamos começar o nosso curso ressaltando um ponto de suma importância: o C é "Case Sensitive", isto é, *maiúsculas e minúsculas fazem diferença*. Se declarar uma variável com o nome soma ela será diferente de **Soma**, **SOMA**, **SoMa** ou **sOmA**. Da mesma maneira, os comandos do C **if** e **for**, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

Dois Primeiros Programas

Vamos ver um primeiro programa em C:

```
#include <stdio.h>
main () /* Um Primeiro Programa */
{
printf ("Ola! Eu estou vivo!\n");
}
```

Compilando e executando este programa você verá que ele coloca a mensagem "Ola! Eu estou vivo!" na tela. Vamos analisar o programa por partes.
A linha **#include <stdio.h>** diz ao compilador que ele deve incluir o arquivo-cabeçalho **stdio.h**. Neste arquivo existem definições de funções úteis. Toda vez que você quiser usar uma destas funções deve-se incluir este comando. O C possui diversos arquivos-cabeçalhos.
Quando fazemos um programa, uma boa idéia é usar comentários que ajudem a elucidar o funcionamento do mesmo. No caso acima temos um comentário: **/* Um Primeiro Programa */**. O compilador C desconsidera qualquer coisa que esteja começando com **/*** e terminando com ***/**. Um comentário pode, inclusive, ter mais de uma linha.
A linha **main()** define uma função de nome **main**. Todos os programas em C têm que ter uma função **main**, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves **{ }**. O código que estiver dentro das chaves será executado seqüencialmente quando a função for chamada.
A única coisa que o programa *realmente* faz é chamar a função **printf()**, passando a string **"Ola! Eu estou vivo!\n"** como argumento. É por causa da função **printf()** que devemos incluir

o arquivo- cabeçalho **stdio.h** . A função **printf()** neste caso irá apenas colocar a string na tela do computador. O **\n** é uma constante chamada de barra invertida. O **n** é de "new line" e ele é interpretado como sendo um retorno de carro. É importante observar também que os *comandos* do C terminam com **;** .

Alguns compiladores C podem dar uma mensagem de aviso ("warning") ao compilar os programas iniciais apresentados aqui. Isto é porque, por default, toda função em C (inclusive a **main()**) retorna um inteiro. Quando não fornecemos este inteiro de retorno, o compilador pode mandar uma mensagem do tipo "Function should return a value.". Por enquanto você terá que aceitar estas mensagens. Mais tarde ensinaremos como devemos fazer para que o programa fique "correto".

Podemos agora tentar um programa mais complicado:

```
#include <stdio.h>
main ()
{
int Dias;                       /* Declaracao de Variaveis */
float Anos;
printf ("Entre com o número de dias: ");/* Entrada de Dados */
scanf ("%d",&Dias);
Anos=Dias/365.25;               /* Conversao Dias->Anos  */
printf ("\n\n%d dias equivalem a %f anos.\n",Dias,Anos);
}
```

Vamos entender como o programa acima funciona. São declaradas duas variáveis chamadas **Dias** e **Anos**. A primeira é um int (inteiro) e a segunda um float (ponto flutuante). É feita então uma chamada à função **printf()**, que coloca uma mensagem na tela.

Queremos agora ler um dado que será fornecido pelo usuário e colocá-lo na variável **Dias**. Para tanto usamos a função **scanf()**. A string **"%d"** diz à função que iremos ler um inteiro. O segundo parâmetro passado à função diz que o dado lido deverá ser armazenado na variável **Dias**. É importante ressaltar a necessidade de se colocar um **&** antes do nome da variável a ser lida quando se usa a função **scanf()**. O motivo disto só ficará claro mais tarde. Observe que, no C, quando temos mais de um parâmetro para uma função, eles serão separados por vírgula.

Temos então uma expressão matemática simples que atribui a **Anos** o valor de **Dias** dividido por 365.25. Como **Anos** é uma variável float o compilador fará uma conversão automática entre os tipos das variáveis.

A segunda chamada à função **printf()** tem três argumentos. A string **"\n\n%d dias eqüivalem a %f anos.\n"** diz à função para dar dois retornos de carro, colocar um inteiro na tela, colocar a mensagem **" dias eqüivalem a "**, colocar um ponto flutuante na tela, colocar a mensagem **" anos."** e dar mais um retorno de carro. Os outros parâmetros são as variáveis das quais devem ser lidos os valores do inteiro e do float, respectivamente.

2.1.1. AUTO AVALIAÇÃO

Veja como você está. O que faz o seguinte programa?

```
#include <stdio.h>
main()
{
int x;
scanf("%d",&x);
printf("%d",x);
}
```

2.2. *Introdução às Funções*

Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O uso de funções permite que o programa fique mais legível, mais bem estruturado. Um programa em C consiste, no fundo, de várias funções colocadas juntas.

Abaixo o tipo mais simples de função:

```
#include <stdio.h>
mensagem ()
{
printf ("Ola! ");
}
main ()
{
mensagem();
printf ("Eu estou vivo!\n");
}
```

Este programa terá o mesmo resultado que o primeiro exemplo da seção anterior. O que ele faz é definir uma função **mensagem()** que coloca uma string na tela. Depois esta função é chamada a partir de **main()** (que também é uma função).

2.2.1. Argumentos

Argumentos são as entradas que a função recebe. É através dos argumentos que passamos parâmetros para a função. Já vimos funções com argumentos. As funções **printf()** e **scanf()** são funções que têm argumentos. Vamos ver um outro exemplo simples de função com argumentos:

```
#include <stdio.h>
square (int x)
{
printf ("O quadrado e %d",(x*x));
}
main ()
{
int num;
printf ("Entre com um numero: ");
scanf ("%d",&num);
printf ("\n\n");
square(num);
}
```

Na definição de **square()** dizemos que a função receberá um argumento inteiro **x**. Quando fazemos a chamada à função, o inteiro **num** é passado como argumento. Há alguns pontos a observar. Em primeiro lugar temos de satisfazer aos requisitos da função quanto ao tipo e à quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo, que o C faz automaticamente, é importante ficar atento. Em segundo lugar, não é importante o nome da variável que se passa como argumento, ou seja, a variável **num**, ao ser passada como argumento para **square()** é copiada para a variável **x**. Dentro de **square()** trabalha-se apenas com **x**. Se mudarmos o valor de **x** dentro de **square()** o valor de **num** na função **main()** permanece inalterado.

Vamos dar um exemplo de função de mais de uma variável. Repare que, neste caso, os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um. Note também que os argumentos passados para a função não necessitam ser todos variáveis porque mesmo sendo constantes serão copiados para a variável de entrada da função.

```
#include <stdio.h>
mult (float a,float b,float c)
{
printf ("%f",a*b*c);
}
main ()
{
float x,y;
x=23.5;
y=12.9;
mult (x,y,3.87);
}
```

2.2.2. Retornando valores

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui retornam um valor inteiro, pois, na linguagem C, a não ser que seja especificado, as funções retornam um inteiro. Mas para dizer ao C *o que* vamos retornar precisamos da palavra reservada **return**. Sabendo disto fica fácil fazer uma função para multiplicar dois inteiros. Veja:

```
#include <stdio.h>
prod (int x,int y)
{
return (x*y);
}
main ()
{
int saida;
saida=prod (12,7);
printf ("A saida e: %d\n",saida);
}
```

Veremos mais adiante como proceder a fim de que uma função retorne outros valores que não sejam inteiros. Quando aprendermos isto poderemos eliminar a mensagem de "warning" mencionada na seção anterior. Note que se você estava recebendo mensagens de "warning" para as funções anteriores, você não recebeu para a função **prod()** do programa anterior! Isto é porque a função **prod()** faz o que o compilador estava esperando: retorna um valor.

```
#include <stdio.h>
float prod (float x,float y)
{
return (x*y);
}
main ()
{
float saida;
saida=prod (45.2,0.0067);
printf ("A saida e: %f\n",saida);
}
```

2.2.3. Forma geral

Apresentamos aqui a forma geral de uma função:

tipo_de_retorno nome_da_função (lista_de_argumentos)
{
código_da_função
}

2.2.4. AUTO AVALIAÇÃO

Veja como você está. Escreva uma função que some dois inteiros e retorne o valor da soma.

2.3. Introdução Básica às Entradas e Saídas

2.3.1. Caracteres

Os caracteres são um tipo de dado: o char. O C trata os caracteres como sendo variáveis de um byte (8 bits). Já os inteiros (ints) têm tem um número maior de bytes. Dependendo da implementação do compilador, eles podem ter 2 bytes (16 bits) ou 4 bytes (32 bits). Isto será melhor explicado na aula 2. Assim sendo, podemos usar um char para armazenar tanto valores numéricos inteiros de 0 a 255 quanto um caractere de texto. Para indicar um caractere de texto usamos apóstrofes. Veja um exemplo de programa que usa caracteres:

```
#include <stdio.h>
main ()
{
char Ch;
Ch='D';
printf ("%c",Ch);
}
```

No programa acima, **%c** indica que **printf()** deve colocar um caractere na tela. Muitas vezes queremos ler um caractere fornecido pelo usuário. Para isto as funções mais usadas, quando se está trabalhando em ambiente DOS ou Windows, são **getch()** e **getche()**. Ambas retornam o caractere pressionado. **getche()** imprime o caractere na tela antes de retorná-lo e **getch()** apenas retorna o caractere pressionado sem imprimi-lo na tela. Ambas as funções podem ser encontradas na biblioteca **conio.h**. Esta biblioteca não está disponível em ambiente Unix (compiladores cc e gcc) e podem, nestes ambientes, ser substituídas pela função scanf(), porém sem as mesmas funcionalidades. Eis um exemplo que usa a função **getch(),** e seu correspondente em ambiente Unix:

```
#include <stdio.h>
#include <conio.h>
main ()
{
char Ch;
Ch=getch();
printf ("Voce pressionou a tecla %c",Ch);
}
```

Equivalente para os compiladores cc e gcc do programa acima, sem usar getch():

```
#include <stdio.h>
main ()
{
char Ch;
scanf("%c", &Ch);
printf ("Voce pressionou a tecla %c",Ch);
}
```

A principal diferença da versão que utiliza getch() para a versão que não utiliza getch() é que no primeiro caso o usuário simplesmente aperta a tecla e o sistema lê diretamente a tecla pressionada. No segundo caso, é necessário apertar também a tecla <ENTER>.

2.3.2. Strings

O Pascal, ao contrário do C, possui um tipo específico para tratar de strings (seqüência de caracteres). No C uma string é um vetor de caracteres terminado com um caractere nulo. O caracter nulo é um caractere com valor igual a zero. O terminador nulo pode ser escrito usando a convenção de barra invertida do C como sendo '\0'. Para declarar uma string podemos usar o seguinte formato geral:

char nome_da_string[tamanho_da_string];

Note que, como temos que reservar um caractere para ser o terminador nulo, temos que declarar o comprimento da string como sendo, no mínimo, um caractere maior que a maior string que pretendemos armazenar. Vamos supor que declaremos uma string de 7 posições e coloquemos a palavra João nela. Teremos:

J	o	A	o	\0	...	...

No caso acima, as duas células não usadas têm valores indeterminados. Isto acontece porque o C *não* inicializa variáveis, cabendo ao programador esta tarefa. Se quisermos ler uma string fornecida pelo usuário podemos usar a função **gets()**. Um exemplo do uso desta função é apresentado abaixo. A função **gets()** coloca o terminador nulo na string, quando você aperta a tecla "Enter".

```
#include <stdio.h>
main ()
{
char string[100];
```

```
printf ("Digite uma string: ");
gets (string);
printf ("\n\nVoce digitou %s",string);
}
```

Como as strings são vetores de caracteres, para se acessar um determinado caracter de uma string, basta "indexarmos", ou seja, usarmos um índice para acessarmos o caracter desejado dentro da string. Suponha uma string chamada *str*. Podemos acessar a segunda letra de *str* da seguinte forma:

```
str[1] = "a";
```

Para isto, basta você lembrar-se que o índice sempre começa em zero. Assim, a primeira letra da string sempre estará na posição 0. A segunda letra sempre estará na posição 1 e assim sucessivamente. Segue um exemplo que imprimirá a segunda letra da string "Joao", apresentada acima. Em seguida, ele mudará esta letra e apresentará a string no final.

```
#include <stdio.h>
main()
{
char str[10] = "Joao";
printf("\n\nString: %s", str);
printf("\nSegunda letra: %c", str[1]);
str[1] = 'U';
printf("\nAgora a segunda letra eh: %c", str[1]);
printf("\n\nString resultante: %s", str);
}
```

Nesta string, o terminador nulo está na posição 4. Das posições 0 a 4, sabemos que temos caracteres válidos, e portanto podemos escrevê-los.
No programa acima, **%s** indica que **printf()** deve colocar uma string na tela. Vamos agora fazer uma abordagem inicial às duas funções que já temos usado para fazer a entrada e saída.

2.3.3. printf

A função **printf()** tem a seguinte forma geral:
printf (string_de_controle,lista_de_argumentos);
Teremos, na string de controle, uma descrição de tudo que a função vai colocar na tela. A string de controle mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os caracteres de controle, que usam a notação **%**. Na string de controle indicamos quais, de qual tipo e em que posição estão as variáveis a serem apresentadas. É muito importante que, para cada caractere de controle, tenhamos um argumento na lista de argumentos. Apresentamos agora alguns dos códigos **%**:

Código	Significado
%d	Inteiro
%f	Float
%c	Caractere
%s	String
%%	Coloca na tela um %

Vamos ver alguns exemplos de **printf()** e o que eles exibem:

```
printf ("Teste %% %%") -> "Teste % %"
printf ("%f",40.345) -> "40.345"
printf ("Um caractere %c e um inteiro %d",'D',120) -> "Um
caractere D e um inteiro 120"
printf ("%s e um exemplo","Este") -> "Este e um exemplo"
printf ("%s%d%%","Juros de ",10) -> "Juros de 10%"
```

Maiores detalhes sobre a função **printf()** serão vistos posteriormente, mas podem ser consultados de antemão pelos interessados.

2.3.4. scanf

O formato geral da função **scanf()** é:

scanf (string-de-controle,lista-de-argumentos);

Usando a função **scanf()** podemos pedir dados ao usuário. Um exemplo de uso, pode ser visto acima. Mais uma vez, devemos ficar atentos a fim de colocar o mesmo número de argumentos que o de caracteres de controle na string de controle. Outra coisa importante é lembrarmos de colocar o **&** antes das variáveis da lista de argumentos. É impossível justificar isto agora, mas veremos depois a razão para este procedimento. Maiores detalhes sobre a função **scanf()** serão vistos posteriormente.

2.3.5. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia duas strings e as coloque na tela. Imprima também a segunda letra de cada string.

2.4. Introdução a Alguns Comandos de Controle de Fluxo

Os comandos de controle de fluxo são aqueles que permitem ao programador alterar a seqüência de execução do programa. Vamos dar uma breve introdução a dois comandos de controle de fluxo.

2.4.1. if

O comando **if** representa uma tomada de decisão do tipo "SE isto ENTÃO aquilo". A sua forma geral é:

if (condição) declaração;

A condição do comando **if** é uma expressão que será avaliada. Se o resultado for zero a declaração não será executada. Se o resultado for qualquer coisa diferente de zero a declaração será executada. A declaração pode ser um bloco de código ou apenas um comando. É interessante notar que, no caso da declaração ser um bloco de código, não é necessário (e nem permitido) o uso do **;** no final do bloco. Isto é uma regra geral para blocos de código. Abaixo apresentamos um exemplo:

```
#include <stdio.h>
main ()
{
int num;
printf ("Digite um numero: ");
scanf ("%d",&num);
if (num>10) printf ("\n\nO numero e maior que 10");
if (num==10)
        {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.");
        }
if (num<10) printf ("\n\nO numero e menor que 10");
}
```

No programa acima a expressão **num>10** é avaliada e retorna um valor diferente de zero, se verdadeira, e zero, se falsa. Repare que quando queremos testar igualdades usamos o operador == e não =. Isto é porque o operador = representa *apenas* uma atribuição. Isto pode parecer estranho à primeira vista, mas se escrevêssemos

```
if (num=10) ...         /* Isto esta errado */
```

o compilador iria *atribuir* o valor 10 à variável **num** e a expressão **num=10** iria retornar 10, fazendo com que o nosso valor de **num** fosse adulterado e fazendo com que a declaração fosse

executada sempre. Este problema gera erros freqüentes entre iniciantes e, portanto, muita atenção deve ser tomada.
Os operadores de comparação são: ==, >, <, >=, <=.

2.4.2. for

O loop (laço) **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:
for (inicialização;condição;incremento) declaração;
O melhor modo de se entender o loop **for** é ver de que maneira ele funciona "por dentro". O loop **for** é equivalente a se fazer o seguinte:

inicialização;

if (condição)

{

declaração;

incremento;

"Volte para o comando if"

}

Podemos ver então que o **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Abaixo vemos um programa que coloca os primeiros 100 números na tela:

```
#include <stdio.h>
main ()
{
int count;
for (count=1;count<=100;count=count+1) printf ("%d ",count);
}
```

2.4.3. AUTO AVALIAÇÃO

Veja como você está. Explique porque está **errado** fazer
if (num=10) ...
O que irá acontecer? Escreva um programa que coloque os números de 1 a 100 na tela na ordem inversa (começando em 100 e terminando em 1).

2.5. Comentários

Como já foi dito, o uso de comentários torna o código do programa mais fácil de se entender. Os comentários do C devem começar com **/*** e terminar com ***/**. O C padrão não permite comentários aninhados (um dentro do outro), mas alguns compiladores os aceitam.

2.5.1. AUTO AVALIAÇÃO

Veja como você está. Escreva comentários para os programas dos exercícios já realizados.

2.6. Palavras Reservadas do C

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser nos seus propósitos originais, isto é, não podemos declarar funções ou variáveis com os mesmos nomes. Como o C é "case sensitive" podemos declarar uma

variável **For**, apesar de haver uma palavra reservada **for**, mas isto não é uma coisa recomendável de se fazer pois pode gerar confusão.
Apresentamos a seguir as palavras reservadas do ANSI C:

auto	double	int	*struct*
break	*else*	long	switch
case	enum	*register*	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

3. AULA 2 - VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

3.1. Nomes de Variáveis

As variáveis no C podem ter qualquer nome se duas condições forem satisfeitas: o nome deve começar com uma letra ou sublinhado (_) e os caracteres subsequentes devem ser letras, números ou sublinhado (_). Há apenas mais duas restrições: o nome de uma variável não pode ser igual a uma palavra reservada, nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C. Variáveis de até 32 caracteres são aceitas. Mais uma coisa: é bom sempre lembrar que o C é "case sensitive" e portanto deve-se prestar atenção às maiúsculas e minúsculas.

3.2. Os Tipos do C

O C tem 5 tipos básicos: **char**, **int**, **float**, **double**, **void**. Destes não vimos ainda os dois últimos. O **double** é o ponto flutuante duplo e pode ser visto como um ponto flutuante com muito mais precisão. O **void** (vazio em inglês) é um tipo especial e deixaremos o seu estudo para mais adiante.
Para cada um dos tipos de variáveis existem os modificadores de tipo. Os modificadores de tipo do C são quatro: **signed**, **unsigned**, **long** e **short**. Ao **float** não se pode aplicar nenhum e ao **double** pode-se aplicar apenas o **long**. Os quatro podem ser aplicados a inteiros. A intenção é que **short** e **long** devam prover tamanhos diferentes de inteiros onde isto for prático. **int** normalmente terá o tamanho natural para uma determinada máquina. Assim, numa máquina de 16 bits, **int** provavelmente terá 16 bits. Numa máquina de 32, **int** deverá ter 32 bits. Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que **shorts** e **ints** devem ocupar pelo menos 16 bits, **longs** pelo menos 32 bits, e **short** não pode ser maior que **int**, que não pode ser maior que **long**. A seguir estão listados os tipos de dados permitidos e seu valores máximos e mínimos em um compilador típico para um hardware de 16 bits:

Tipo	Num de bits	Intervalo	
		Inicio	Fim
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	16	-32.768	32.767
unsigned int	16	0	65.535
signed int	16	-32.768	32.767
short int	16	-32.768	32.767
unsigned short int	16	0	65.535
signed short int	16	-32.768	32.767
long int	32	-2.147.483.648	2.147.483.647

signed long int	32	-2.147.483.648	2.147.483.647
unsigned long int	32	0	4.294.967.295
float	32	3,4E-38	3.4E+38
double	64	1,7E-308	1,7E+308
Long double	80	3,4E-4932	3,4E+4932

O tipo **long double** é o tipo de ponto flutuante com maior precisão. É importante observar que os intervalos de ponto flutuante, na tabela acima, estão indicados em faixa de *expoente*, mas os números podem assumir valores tanto positivos quanto negativos.

3.3. Declaração e Inicialização de Variáveis

As variáveis no C devem ser declaradas antes de serem usadas. A forma geral da declaração de variáveis é:

tipo_da_variável lista_de_variáveis;

As variáveis da lista de variáveis terão todas o mesmo tipo e deverão ser separadas por vírgula. Como o tipo default do C é o int, quando vamos declarar variáveis int com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo. Assim um long basta para declarar um **long int**.

Por exemplo, as declarações

```
char ch, letra;
long count;
float pi;
```

declaram duas variáveis do tipo **char** (ch e letra), uma variável **long int** (count) e um **float** pi.

Há três lugares nos quais podemos declarar variáveis. O primeiro é fora de todas as funções do programa. Estas variáveis são chamadas variáveis globais e podem ser usadas a partir de qualquer lugar no programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as vêem. O segundo lugar no qual se pode declarar variáveis é no início de um bloco de código de uma função. Estas variáveis são chamadas locais e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável. O terceiro lugar onde se pode declarar variáveis é na lista de parâmetros de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas.

Veja o programa abaixo:

```
#include <stdio.h>
int contador;
main()
{
  char condicao;
  for (int i=0; ...)  /* loop 1 */
  {
    ...
  }
  ...
}
```

A variável *contador* e uma variável global, e é acessível de qualquer parte do programa. A variável *condição,* só existe dentro de main(). A variável inteira *i* é um exemplo de declaração na lista de parâmetros de uma função.

As regras que regem *onde* uma variável é válida chamam-se regras de *escopo* da variável. Há mais dois detalhes que devem ser ressaltados. Duas variáveis globais não podem ter o mesmo nome. O mesmo vale para duas variáveis locais de uma mesma função. Já duas variáveis locais, de funções diferentes, podem ter o mesmo nome sem perigo algum de conflito.

Podemos inicializar variáveis no momento de sua declaração. Para fazer isto podemos usar a forma geral

tipo_da_variável nome_da_variável = constante;

Isto é importante pois quando o C cria uma variável ele *não* a inicializa. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor *indefinido* e que não pode

ser utilizado para nada. *Nunca* presuma que uma variável declarada vale zero ou qualquer outro valor. Exemplos de inicialização são dados abaixo:

```
char ch='D';
int count=0;
float pi=3.141;
```

3.3.1. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia uma string (use gets()) e retire todos os caracteres 'c' da string lida. Informe na tela quantos caracteres foram retirados.

3.4. Constantes

Constantes são valores que são mantidos fixos pelo compilador. Já usamos constantes neste curso. São consideradas constantes, por exemplo, os números e caracteres como 45.65 ou 'n', Tc...

3.4.1. Constantes dos tipos básicos

Abaixo vemos as constantes relativas aos tipos básicos do C:

Tipo de Dados	Exemplos de Constantes
char	'b' '\n' '\0'
Int	2 32000 –130
long int	100000 -467
short int	100 -30
Unsigned int	50000 35678
Float	0.0 23.7 -12.3e-10
Double	12546354334.0 -0.0000034236556

3.4.2. Constantes hexadecimais e octais

Muitas vezes precisamos inserir constantes hexadecimais (base dezesseis) ou octais (base oito) no nosso programa. O C permite que se faça isto. As constantes hexadecimais começam com 0x. As constantes octais começam em 0. Alguns exemplos:

Constante	Tipo
0xEF	Char Hexadecimal (8 bits)
0x12A4	Int Hexadecimal (16 bits)
03212	Char Octal (8 bits)
034215432	Int Octal (16 bits)

Nunca escreva portanto 013 achando que o C vai compilar isto como se fosse 13. Na linguagem C 013 é diferente de 13!

3.4.3. Constantes strings

Já mostramos como o C trata strings. Vamos agora alertar para o fato de que uma string **"Joao"** é na realidade uma constante string. Isto implica, por exemplo, no fato de que **'t'** é diferente de **"t"**, pois **'t'** é um char enquanto que **"t"** é um vetor de dois chars onde o primeiro é **'t'** e o segundo é **'\0'**.

3.4.4. Constantes de barra invertida

O C utiliza, para nos facilitar a tarefa de programar, vários códigos chamados códigos de barra invertida. Estes são caracteres que podem ser usados como qualquer outro. A lista completa dos códigos de barra invertida é dada a seguir:

Código	Significado
\b	Retrocesso ("back")
\f	Alimentação de formulário ("form feed")
\n	Nova linha ("new line")
\r	Retorno de carro ("carriage return")
\t	Tabulação horizontal ("tab")
\"	Aspas
\'	Apóstrofo
\0	Nulo (0 em decimal)
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro ("beep")

3.5. Operadores Aritméticos e de Atribuição

Os operadores aritméticos são usados para desenvolver operações matemáticas. A seguir apresentamos a lista dos operadores aritméticos do C:

Operador	Ação
+	Soma (inteira e ponto flutuante)
-	Subtração ou Troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
--	Decremento (inteiro e ponto flutuante)

O C possui operadores unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma é um operador binário pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores **-** (subtração), *****, **/** e **%**. O operador **-** como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por -1.
Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar, a variável sobre a qual estão aplicados, de 1. Então

```
x++;
x--;
```

são equivalentes a

```
x=x+1;
x=x-1;
```

Estes operadores podem ser pré-fixados ou pós- fixados. A diferença é que quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Então, em

```
x=23;
y=x++;
```

teremos, no final, **y=23** e **x=24**. Em

```
x=23;
y=++x;
```

teremos, no final, **y=24** e **x=24**. Uma curiosidade: a linguagem de programação C++ tem este nome pois ela seria um "incremento" da linguagem C padrão. A linguagem C++ é igual a linguagem C só que com extensões que permitem a programação orientada a objeto, o que é um recurso extra.
O operador de atribuição do C é o =. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto ele retorna o valor que ele atribuiu. Isto faz com que as seguintes expressões sejam válidas:

```
x=y=z=1.5;            /* Expressao 1 */

if (k=w) ...    /* Expressao 2 */
```

A expressão 1 é válida, pois quando fazemos **z=1.5** ela retorna 1.5, que é passado adiante. A expressão dois será verdadeira se **w** for diferente de zero, pois este será o valor retornado por **k=w**. Pense bem antes de usar a expressão dois, pois ela pode gerar erros de interpretação. Você *não* está comparando **k** e **w**. Você está atribuindo o valor de **w** a **k** e usando este valor para tomar a decisão.

3.5.1. AUTO AVALIAÇÃO

Veja como você está. Diga o resultado das variáveis x, y e z depois da seguinte seqüência de operações:

```
int x,y,z;
x=y=10;
z=++x;
x=-x;
y++;
x=x+y-(z--);
```

3.6. Operadores Relacionais e Lógicos

Os operadores relacionais do C realizam ***comparações*** entre variáveis. São eles:

Operador	Ação
>	Maior do que
>=	Maior ou igual a
<	Menor do que
<=	Menor ou igual a
==	Igual a
!=	Diferente de

Os operadores relacionais retornam verdadeiro (1) ou falso (0). Para fazer ***operações com valores lógicos*** (verdadeiro e falso) temos ***os operadores lógicos***:

Operador	Ação
&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

Usando os operadores relacionais e lógicos podemos realizar uma grande gama de testes. A tabela-verdade destes operadores é dada a seguir:

p	q	p AND q	p or q

falso	falso	falso	falso
falso	verdadeiro	falso	verdadeiro
verdadeiro	falso	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro	verdadeiro

3.7. Operadores Lógicos Bit a Bit

O C permite que se faça ***operações lógicas "bit-a- bit"*** em números. Esta característica ajuda programadores que queiram trabalhar com o computador em "baixo nível". As operações lógicas só podem ser usadas nos tipos **char**, **int** e **long int**. Os operadores são:

Operador	Ação
&	AND
|	OR
^	XOR (OR exclusivo)
~	NOT
>>	Deslocamento de bits a direita
<<	Deslocamento de bits a esquerda

Os operadores **&**, |, ^ e ~ são as operações lógicas bit a bit. A forma geral dos operadores de deslocamento é:

valor>>número_de_deslocamentos
valor<<número_de_deslocamentos

O número_de_deslocamentos indica o quanto cada bit irá ser deslocado.

3.7.1. AUTO AVALIAÇÃO

Veja como você está. Diga se as seguintes expressões serão verdadeiras ou falsas:

-> ((10>5)||(5>10))
-> (!(5==6)&&(5!=6)&&((2>1)||(5<=4)))

3.8. Expressões

Expressões são combinações de variáveis, constantes e operadores. Quando montamos expressões temos que levar em consideração a ordem com que os operadores são executados, conforme a tabela de precedências da linguagem C.

Exemplos de expressões:

```
Anos=Dias/365.25;
i = i+3;
c= a*b + d/e;
c= a*(b+d)/e;
```

3.8.1. Conversão de tipos em expressões

Quando o C avalia expressões onde temos variáveis de tipos diferentes o compilador verifica se as conversões são possíveis. Se não são, ele não compilará o programa, dando uma mensagem de erro. Se as conversões forem possíveis ele as faz, seguindo as regras abaixo:

1. Todos os chars e short ints são convertidos para ints. Todos os floats são convertidos para doubles.
2. Para pares de operandos de tipos diferentes: se um deles é long double o outro é convertido para long double; se um deles é double o outro é convertido para double; se um é long o outro é convertido para long; se um é unsigned o outro é convertido para unsigned.

3.8.2. Expressões que Podem ser Abreviadas

O C admite as seguintes equivalência, que podem ser usadas para simplificar expressões ou para facilitar o entendimento de um programa:

Expressão Original	Expressão Equivalente
x=x+k;	x+=k;
x=x-k;	x-=k;
x=x*k;	x*=k;
x=x/k;	x/=k;
x=x>>k;	x>>=k;
x=x<<k;	x<<=k;
x=x&k;	x&=k;
Tc...	

3.8.3. Encadeando expressões: o operador ,

O operador **,** determina uma lista de expressões que devem ser executadas seqüencialmente. O valor retornado por uma expressão com o operador , é sempre dado pela expressão mais à direita. No exemplo abaixo:

```
x=(a=2,a+3);
```

a variável **x** receberá o valor 5. Pode-se encadear quantos operadores , forem necessários.

3.8.4. Tabela de Precedências do C

Esta é a tabela de precedência dos operadores em C. Alguns (poucos) operadores ainda não foram estudados, e serão apresentados em aulas posteriores.

<table>
<tr><th>Mais Baixa</th><td>() [] -></td></tr>
<tr><td></td><td>! ~ ++ -- . -(unário) (cast) *(unário) &(unário) sizeof</td></tr>
<tr><td></td><td>* / %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><< >></td></tr>
<tr><td></td><td><<= >>=</td></tr>
<tr><td></td><td>== !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td></td><td>?</td></tr>
<tr><td></td><td>= += -= *= /=</td></tr>
<tr><th>Mais alta</th><td>,</td></tr>
</table>

Uma dica aos iniciantes: Você não precisa saber toda a tabela de precedências de cor. E útil que você conheça as principais relações, mas é aconselhável que ao escrever o seu código, você tente isolar as expressões com parênteses, para tornar o seu programa mais legível.

3.9. Modeladores (Casts)

Um modelador é aplicado a uma expressão. Ele *força* a mesma a ser de um tipo especificado. Sua forma geral é:

(tipo)expressão

Um exemplo:

```
#include <stdio.h>
main ()
{
int num;
float f;
```

```
num=10;
f=(float)num/7;
printf ("%f",f);
}
```

Se não tivéssemos usado o modelador no exemplo acima o C faria uma divisão inteira entre 10 e 7. O resultado seria um e este seria depois convertido para float mas continuaria a ser 1.0. Com o modelador temos o resultado correto.

3.9.1. AUTO AVALIAÇÃO

Veja como você está. Compile o exemplo acima sem usar o modelador, e verifique os resultados. Compile-o novamente usando o modelador e compare a saída com os resultados anteriores.

4. AULA 3 - ESTRUTURAS DE CONTROLE DE FLUXO

As estruturas de controle de fluxo são fundamentais para qualquer linguagem de programação. Sem elas só haveria uma maneira do programa ser executado: de cima para baixo comando por comando. Não haveria condições, repetições ou saltos. A linguagem C possui diversos comandos de controle de fluxo. É possível resolver todos os problemas sem utilizar todas elas, mas devemos nos lembrar que a elegância e facilidade de entendimento de um programa dependem do uso correto das estruturas no local certo.

4.1. O Comando if

Já introduzimos o comando **if**. Sua forma geral é:

if (condição) declaração;

A expressão, na condição, será avaliada. Se ela for zero, a declaração não será executada. Se a condição for diferente de zero a declaração será executada. Aqui reapresentamos o exemplo de um uso do comando **if**:

```
#include <stdio.h>
main ()
{
int num;
printf ("Digite um numero: ");
scanf ("%d",&num);
if (num>10)
   printf ("\n\nO numero e maior que 10");
if (num==10)
        {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.");
        }
if (num<10)
   printf ("\n\nO numero e menor que 10");
}
```

4.2. O else

Podemos pensar no comando **else** como sendo um complemento do comando if. O comando if completo tem a seguinte forma geral:

if (condição) declaração_1;
else declaração_2;

A expressão da condição será avaliada. Se ela for diferente de zero a declaração 1 será executada. Se for zero a declaração 2 será executada. É importante nunca esquecer que, quando

usamos a estrutura **if-else**, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Abaixo está um exemplo do uso do **if-else** que deve funcionar como o programa da seção anterior.

```
#include <stdio.h>
            main ()
            {
            int num;
            printf ("Digite um numero: ");
            scanf ("%d",&num);
            if (num==10)
                  {
                  printf ("\n\nVoce acertou!\n");
                  printf ("O numero e igual a 10.\n");
                  }
            else
                  {
                  printf ("\n\nVoce errou!\n");
                  printf ("O numero e diferente de 10.\n");
                  }
            }
```

4.3. O if-else-if

A estrutura **if-else-if** é apenas uma extensão da estrutura if-else. Sua forma geral pode ser escrita como sendo:

if (condição_1) declaração_1;
else if (condição_2) declaração_2;
else if (condição_3) declaração_3;
.
.
.
else if (condição_n) declaração_n;
else declaração_default;

A estrutura acima funciona da seguinte maneira: o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero. Neste caso ele executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à *primeira* condição que der diferente de zero. A última declaração (default) é a que será executada no caso de todas as condições darem zero e é opcional. Um exemplo da estrutura acima:

```
#include <stdio.h>
main ()
{
int num;
printf ("Digite um numero: ");
scanf ("%d",&num);
if (num>10)
     printf ("\n\nO numero e maior que 10");
else if (num==10)
        {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.");
        }
     else if (num<10)
             printf ("\n\nO numero e menor que 10");
}
```

4.4. A expressão condicional

Quando o compilador avalia uma condição, ele quer um valor de retorno para poder tomar a decisão. Mas esta expressão não necessita ser uma expressão no sentido convencional. Uma variável sozinha pode ser uma "expressão" e esta retorna o seu próprio valor. Isto quer dizer que teremos as seguintes equivalência:

```
int num;
if (num!=0) ....
if (num==0) ....
```

eqüivale a

```
int num;
if (num) ....
if (!num) ....
```

Isto quer dizer que podemos simplificar algumas expressões simples.

4.5. ifs aninhados

O if aninhado é simplesmente um if dentro da declaração de um outro if externo. O único cuidado que devemos ter é o de saber exatamente a qual if um determinado else está ligado. Vejamos um exemplo:

```
#include <stdio.h>
main ()
{
int num;
printf ("Digite um numero: ");
scanf ("%d",&num);
if (num==10)
        {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.\n");
        }
else
        {
        if (num>10)
                {
                printf ("O numero e maior que 10.");
                }
        else
                {
                printf ("O numero e menor que 10.");
                }
        }
}
```

4.6. O Operador ?

Uma expressão como:

```
if (a>0)
        b=-150;
else
        b=150;
```

pode ser simplificada usando-se o operador **?** da seguinte maneira:

```
b=a>0?-150:150;
```

De uma maneira geral expressões do tipo:

if (condição)

expressão_1;

else

expressão_2;

podem ser substituídas por:

condição?expressão_1:expressão_2;

O operador **?** é limitado (não atende a uma gama muito grande de casos) mas pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular. Veja o exemplo:

```
#include <stdio.h>
main()
{
     int index = 0, contador;
     char letras[5] = "Joao";
     for (contador=0; contador<1000; contador++)
        {
        printf("\n%c",letras[index]);
        index = (index == 4)? index=0: ++index;
        }
}
```

O nome Joao é escrito na tela verticalmente até a variável contador determinar o término do programa. Enquanto isto a variável index assume os valores 0, 1, 2, 3, 4, 0, 1, ... progressivamente.

4.6.1. AUTO-AVALIAÇÃO

Veja como você está. Altere o último exemplo para que ele escreva cada letra 5 vezes seguidas. Para isto, use um 'if' para testar se o contador é divisível por cinco (utilize o operador %) e só então realizar a atualização em index.

4.7. O Comando switch

O comando if-else e o comando **switch** são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o if, mas o comando **switch** tem aplicações valiosas. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando **switch** é próprio para se testar uma variável em relação a diversos valores preestabelecidos. Sua forma geral é:

```
switch (variável)
   {
   case constante_1:
   declaração_1;
   break;
   case constante_2:
   declaração_2;
   break;
   .
   .
   .
   case constante_n:
   decalração_n;
   break;
   default
   declaração_default;
   }
```

Podemos fazer uma analogia entre o **switch** e a estrutura if-else-if apresentada anteriormente. A diferença fundamental é que a estrutura **switch** *não* aceita expressões. Aceita apenas constantes. O **switch** testa a variável e executa a declaração cujo **case** corresponda ao valor atual da variável. A declaração **default** é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.
O comando break, faz com que o **switch** seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando **switch**. Se após a execução da declaração não houver um break, o programa continuará executando. Isto pode ser útil em algumas situações, mas eu recomendo cuidado. Veremos agora um exemplo do comando **switch**:

```
#include <stdio.h>
main ()
{
int num;
printf ("Digite um numero: ");
scanf ("%d",&num);
switch (num)
        {
        case 9:
                printf ("\n\nO numero e igual a 9.\n");
        break;
        case 10:
                printf ("\n\nO numero e igual a 10.\n");
        break;
        case 11:
                printf ("\n\nO numero e igual a 11.\n");
        break;
        default:
                printf ("\n\nO numero nao e nem 9 nem 10 nem
11.\n");
        }
}
```

4.7.1. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa utilizando o comando *switch* que leia uma string (use gets()) e substitua todos os espaços e tabulações ('\t') por caracteres de nova linha. O loop deve encerrar quando encontrar o caracter de final de linha '\0'.

4.8. O Comando for

O **for** é o primeiro de uma série de três comandos para se trabalhar com loops. Os outros são o while e o do. Os três compõem a segunda família de comandos de controle de fluxo. Podemos pensar nesta família como sendo a dos comandos de repetição controlada.
Como já foi dito, o loop **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:
for (inicialização;condição;incremento) declaração;
O melhor modo de se entender o loop **for** é ver como ele funciona "por dentro". O loop **for** é equivalente a se fazer o seguinte:

inicialização;

if (condição)

{
declaração;
incremento;
"Volte para o comando if"
}

Podemos ver, então, que o **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Um ponto importante é que podemos omitir qualquer um dos elementos do **for**, isto é, se não quisermos uma inicialização poderemos omiti-la. Abaixo vemos um programa que coloca os primeiros 100 números inteiros na tela:

```
#include <stdio.h>

main ()
{
int count;
for (count=1;count<=100;count++) printf ("%d ",count);
}
```

Note que, no exemplo acima, há uma diferença em relação ao exemplo anterior. O incremento da variável **count** é feito usando o operador de incremento que nós agora já conhecemos. Esta é a forma usual de se fazer o incremento (ou decremento) em um loop **for**.
Todas as linguagens de programação possuem algo parecido com o **for** do C. Mas, o **for** na linguagem C é muito mais flexível e poderoso. Temos acesso à inicialização, à condição e ao incremento. Isto nos permite fazer o que quisermos. Este é o poder do C.

4.8.1. O loop infinito

O loop infinito tem a forma

for (;;) declaração;

Este loop chama-se loop infinito porque será executado para sempre, a não ser que ele seja interrompido. Para interromper um loop como este usamos o comando break. O comando break vai quebrar o loop infinito e o programa continuará sua execução normalmente.
Como exemplo vamos ver um programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuário aperte uma tecla especial, denominada FLAG. O nosso FLAG será a letra 'X'.

```
#include <stdio.h>

main ()
{
int Count;
char ch;
for (Count=1;;Count++)
        {
        fflush(NULL);
        scanf("%c",&ch);
        if (ch == 'X') break;
        printf("\nLetra: %c\n",ch);
        }
}
```

Atenção ao comando *fflush(NULL)*. O papel deste comando é limpar o buffer do teclado para que outros caracteres armazenados no buffer do computador sejam liberados. Desta forma a leitura de caracter que acontece logo após a sua execução não ficará prejudicada.

4.8.2. O loop sem conteúdo

Loop sem conteúdo é aquele no qual se omite a declaração. Sua forma geral é portanto (atenção ao ponto e vírgula!):

for (inicialização;condição;incremento);

Uma das aplicações desta estrutura é gerar tempos de espera. O programa

```
#include <stdio.h>

main ()
{
long int i;
```

```
printf("\a");                  /* Imprime o caracter de alerta
(um beep) */
for (i=0; i<10000000; i++);   /* Espera 10.000.000 de iteracoes
*/
printf("\a");                  /* Imprime outro caracter de
alerta */
}
```
demonstra o espaço de tempo gerado.

4.8.3. AUTO AVALIAÇÃO

Veja como você está. Faça um programa que inverta uma string (uma seqüência de caracteres). Use o comando for para varrer a string até o seu final. Lembre-se que a string sempre termina com o caracter '\0'.

4.9. O Comando while

O comando **while** tem a seguinte forma geral:

while (condição) declaração;

Assim como fizemos para o comando anterior, vamos tentar mostrar como o **while** funciona fazendo uma analogia. Então o **while** seria:

if (condição)

{
declaração;
"Volte para o comando if"
}

Podemos ver que a estrutura **while** testa uma condição. Se esta for verdadeira a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do for, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo. Vamos ver um exemplo do uso do **while**. O programa abaixo espera que usuário digitar a tecla 'q' e só depois finaliza:

```
#include <stdio.h>
main ()
{
char Ch;
Ch='\0';
while (Ch!='q')
       {
       fflush(NULL);
       scanf("%c",&Ch);
       }
}
```

4.9.1. AUTO AVALIAÇÃO

Veja como você está. Refaça o programa da página anterior. Use o comando *while* para fechar o loop.

4.10. O Comando do

O terceiro loop que veremos é o **do-while** de forma geral:

do

{
declaração;
} while (condição);

Mesmo que a declaração seja apenas um comando é uma boa prática deixar as chaves. O ponto-e-vírgula final é obrigatório. Vamos, como anteriormente, ver o funcionamento da estrutura **do-while** "por dentro":

declaração;
if (condição) "Volta para a declaração"

Vemos pela análise do bloco acima que a estrutura **do-while** executa a declaração, testa a condição e, se esta for verdadeira, volta para a declaração. A grande novidade no comando **do-while** é que ele, ao contrário do for e do while, garante que a declaração será executada pelo menos uma vez.

Um dos principais usos da estrutura **do-while** é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido. Um caso está exemplificado abaixo:

```
#include <stdio.h>
main ()
{
char Ch;
do
        {
        printf ("\n\nEscolha um:\n\n");
        printf ("\t(1)...Mamao\n");
        printf ("\t(2)...Abacaxi\n");
        printf ("\t(3)...Laranja\n\n");
        fflush(NULL);
        scanf("%c",&Ch);
        } while ((Ch!='1')&&(Ch!='2')&&(Ch!='3'));
switch (Ch)
        {
        case 1:
                printf ("\t\tVoce escolheu Mamao.\n");
        break;
        case 2:
                printf ("\t\tVoce escolheu Abacaxi.\n");
        break;
        case 3:
                printf ("\t\tVoce escolheu Laranja.\n");
        break;
        }
}
```

4.10.1.AUTO AVALIAÇÃO

Veja como você está. Refaça o exercício da página c410.html utilizando o laço do-while para controlar o fluxo.

4.11. O Comando break

Nós já vimos dois usos para o comando **break**. Ele interrompe os comandos switch e for. Na verdade estes são os dois usos do comando **break**: ele quebra a execução do comando **break** e interrompe a execução de *qualquer* loop. O **break** faz com que a execução do programa continue na primeira linha seguinte ao loop.

4.12. O Comando continue

O comando **continue** pode ser visto como sendo o oposto do break. Ele só funciona dentro de um loop. Quando o comando **continue** é encontrado, o loop pula para a próxima iteração.
O programa abaixo exemplifica o uso do *continue*:

```
#include <stdio.h>
main()
{
int opcao;
while (opcao != 5)
       {
       printf("\n\n Escolha uma opcao entre 1 e 5: ");
       scanf("%d", &opcao);
       if ((opcao > 5)||(opcao <1)) continue;  /* Opcao
invalida: volta ao inicio do loop */
       switch (opcao)
          {
          case 1:
               printf("\n --> Primeira opcao..");
          break;
          case 2:
               printf("\n --> Segunda opcao..");
          break;
          case 3:
               printf("\n --> Terceira opcao..");
          break;
          case 4:
               printf("\n --> Quarta opcao..");
          break;
          case 5:
               printf("\n --> Abandonando..");
          break;
          }
       }
}
```

O programa acima ilustra uma simples e útil aplicação para o *continue*. Ele recebe uma opção do usuário. Se esta opção for inválida, o *continue* faz com que o fluxo seja desviado de volta ao início do loop. Caso a opção escolhida seja válida o programa segue normalmente.

4.13. O Comando goto

O **goto** é o último comando de controle de fluxo. Ele pertence a uma classe à parte: a dos comandos de salto incondicional. O **goto** realiza um salto para um local especificado. Este local é determinado por um rótulo. Um rótulo, na linguagem C, é uma marca no programa. Você dá o nome que quiser a esta marca. Podemos tentar escrever uma forma geral:

nome_do_rótulo:

....
goto nome_do_rótulo;
....

Devemos declarar o nome do rótulo na posição para a qual vamos dar o salto seguido de **:**. O **goto** pode saltar para um rótulo que esteja mais à frente ou para trás no programa. Uma observação importante é que o rótulo e o **goto** devem estar dentro da mesma *função*. Como exemplo do uso do **goto** vamos rescrever o equivalente ao comando for apresentado na seção equivalente ao mesmo:

inicialização;
início_do_loop:
if (condição)

{
declaração;
incremento;
goto início_do_loop;
}

O comando **goto** deve ser utilizado com parcimônia, pois o abuso no seu uso tende a tornar o código confuso. O **goto** não é um comando *necessário*, mas o seu bom emprego pode facilitar o entendimento de algumas funções. Há quem diga que o **goto** nunca deve ser usado, mas isto não é verdade. Isto é um extremismo que é em muito devido à linguagem BASIC que abusa dos **goto**s e não tem estruturas ricas como as do C. O comando **goto** pode tornar um código muito mais fácil de se entender *se* ele for bem empregado. Um caso em que ele pode ser útil é quando temos vários loops e ifs aninhados e se queira, por algum motivo, sair destes loops e ifs todos de uma vez. Neste caso um **goto** resolve o problema muito mais elegantemente que vários breaks, sem contar que os breaks exigiriam muito mais testes. Ou seja, neste caso o **goto** é mais elegante e mais rápido. Mas não abuse.

O exemplo da página anterior pode ser rescrito usando-se o *goto*:

```
#include <stdio.h>
main()
{
int opcao;
while (opcao != 5)
       {
REFAZ: printf("\n\n Escolha uma opcao entre 1 e 5: ");
       scanf("%d", &opcao);
       if ((opcao > 5)||(opcao <1)) goto REFAZ;  /* Opcao
invalida: volta
                                                    ao rotulo REFAZ
*/
       switch (opcao)
          {
          case 1:
               printf("\n --> Primeira opcao..");
          break;
          case 2:
               printf("\n --> Segunda opcao..");
          break;
          case 3:
               printf("\n --> Terceira opcao..");
          break;
          case 4:
               printf("\n --> Quarta opcao..");
          break;
          case 5:
               printf("\n --> Abandonando..");
          break;
          }
       }
}
```

4.13.1.AUTO AVALIAÇÃO

Veja como você está.

Escreva um programa que peça três inteiros, correspondentes a dia , mês e ano. Peça os números até conseguir valores que estejam na faixa correta (dias entre 1 e 31, mês entre 1 e 12 e ano entre 1900 e 2100). Verifique se o mês e o número de dias batem (incluindo verificação de anos bissextos). Se estiver tudo certo imprima o número que aquele dia corresponde no ano. Comente seu programa. Faça as indentações corretamente.

Ps: Um ano é bissexto se for divisível por 4 e não for divisível por 100, exceto para os anos divisíveis por 400, que também são bissextos.

5. AULA 4 - MATRIZES E STRINGS

5.1. Vetores

Vetores nada mais são que matrizes unidimensionais. Vetores são uma estrutura de dados muito utilizada. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Para se declarar um vetor podemos utilizar a seguinte forma geral:

tipo_da_variável nome_da_variável [tamanho];

Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

```
float exemplo [20];
```

o C irá reservar 4x20=80 bytes. Estes bytes são reservados de maneira contígua. Na linguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

```
exemplo[0]
exemplo[1]
.
.
.
exemplo[19]
```

Mas ninguém o impede de escrever:

```
exemplo[30]
exemplo[103]
```

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que *você* deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter sobrescritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
#include <stdio.h>
main ()
{
int num[100];  /* Declara um vetor de inteiros de 100 posicoes
*/
int count=0;
int totalnums;
do
        {
        printf ("\nEntre com um numero (-999 p/ terminar): ");
        scanf ("%d",&num[count]);
        count++;
        } while (num[count-1]!=-999);
totalnums=count-1;
printf ("\n\n\n\t Os números que você digitou foram:\n\n");
for (count=0;count<totalnums;count++)
        {
        printf (" %d",num[count]);
        }
}
```

No exemplo acima, o inteiro *count* é inicializado em 0. O programa pede pela entrada de números até que o usuário entre com o Flag -999. Os números são armazenados no vetor **num**. A cada número armazenado, o contador do vetor é incrementado para na próxima iteração escrever na próxima posição do vetor. Quando o usuário digita o flag, o programa abandona o primeiro loop e armazena o total de números gravados. Por fim, todos os números são impressos. É bom lembrar aqui que nenhuma restrição é feita quanto a quantidade de números

digitados. Se o usuário digitar mais de 100 números, o programa tentará ler normalmente, mas o programa os escreverá em uma parte não alocada de memória, pois o espaço alocado foi para somente 100 inteiros. Isto pode resultar nos mais variados erros, principalmente em tempo de execução.

5.1.1. AUTO AVALIAÇÃO

Veja como você está. Rescreva o exemplo acima, realizando a cada leitura um teste para ver se a dimensão do vetor não foi ultrapassada. Caso o usuário entre com 100 números, o programa deverá abortar o loop de leitura automaticamente. O uso do Flag não deve ser retirado.

5.2. Strings

Strings são vetores de chars. Nada mais e nada menos. As strings são o uso mais comum para os vetores. Devemos apenas ficar sempre atentos para o fato de que as strings têm o seu último elemento como sendo um '\0'. A declaração geral para uma string é:

char nome_da_string [tamanho];

Devemos lembrar que o tamanho da string deve incluir o '\0' final. A biblioteca do C possui diversas funções que manipulam strings. Estas funções são úteis pois não se pode, por exemplo, igualar duas strings:

```
string1=string2;        /* NAO faca isto */
```

Fazer isto é um desastre. Quando você terminar de ler a seção que trata de ponteiros você entenderá porquê. As strings devem ser igualadas elemento a elemento.

Quando vamos fazer programas que tratam de string muitas vezes podemos fazer bom proveito do fato de que uma string termina com '\0'. Veja, por exemplo, o programa abaixo que serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):

```
#include <stdio.h>
main ()
{
int count;
char str1[100],str2[100];
....            /* Aqui o programa le str1 que sera copiada
para str2 */
for (count=0;str1[count];count++)
        {
        str2[count]=str1[count];
        }
str2[count]='\0';
....            /* Aqui o programa continua */
}
```

A condição no loop for acima é baseada no fato de que a string que está sendo copiada termina em '\0'. Este tipo de raciocínio é a base do C e você deve fazer um esforço para entender como é que o programa acima funciona. Quando o elemento encontrado em **str1[count]** é o '\0', o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

Vamos ver agora algumas funções básicas para manipulação de strings.

5.2.1. gets

A função **gets()** lê uma string do teclado. Sua forma geral é:

gets (nome_da_string);

O programa abaixo demonstra o funcionamento da função **gets()**:

```
#include <stdio.h>
main ()
{
char string[100];
printf ("Digite o seu nome: ");
```

```
gets (string);
printf ("\n\n Ola %s",string);
}
```

Repare que é válido passar para a função **printf()** o nome da string. Você verá mais adiante porque isto é válido. Como o primeiro argumento da função **printf()** é uma string também é válido fazer:

```
printf (string);
```

5.2.2. strcpy

Sua forma geral é:

strcpy (string_destino,string_origem);

A função **strcpy()** copia a string-origem para a string- destino. Seu funcionamento é semelhante ao da rotina apresentada na seção anterior. As funções apresentadas nestas seções estão na biblioteca do C **string.h**. A seguir apresentamos um exemplo de uso da função **strcpy()**:

```
#include <stdio.h>
#include <string.h>
main ()
{
char str1[100],str2[100],str3[100];
printf ("Entre com uma string: ");
gets (str1);
strcpy (str2,str1);
strcpy (str3,"Voce digitou a string ");
printf ("\n\n%s%s",str3,str2);
}
```

5.2.3. strcat

A função **strcat()** tem a seguinte forma geral:

strcat (string_destino,string_origem);

A string de origem permanecerá inalterada e será anexada ao fim da string de destino. Um exemplo:

```
#include <stdio.h>
#include <string.h>
main ()
{
char str1[100],str2[100];
printf ("Entre com uma string: ");
gets (str1);
strcpy (str2,"Voce digitou a string ");
strcat (str2,str1);
printf ("\n\n%s",str2);
}
```

5.2.4. strlen

Sua forma geral é:

strlen (string);

A função **strlen()** retorna o comprimento da string fornecida. O terminador nulo não é contado. Isto quer dizer que, de fato, o comprimento do vetor da string deve ser um a mais que o inteiro retornado por **strlen()**. Um exemplo do seu uso:

```
#include <stdio.h>
#include <string.h>
main ()
{
int size;
char str[100];
```

```
printf ("Entre com uma string: ");
gets (str);
size=strlen (str);
printf ("\n\nA string que voce digitou tem tamanho %d",size);
}
```

5.2.5. strcmp

Sua forma geral é:

strcmp (string1,string2);

A função **strcmp()** compara a string 1 com a string 2. Se as duas forem idênticas a função retorna zero. Se elas forem diferentes a função retorna não-zero. Um exemplo da sua utilização:

```
#include <stdio.h>
#include <string.h>
main ()
{
char str1[100],str2[100];
printf ("Entre com uma string: ");
gets (str1);
printf ("\n\nEntre com outra string: ");
gets (str2);
if (strcmp(str1,str2))
        printf ("\n\nAs duas strings são diferentes.");
else printf ("\n\nAs duas strings são iguais.");
}
```

5.2.6. AUTO AVALIAÇÃO

Veja como você está. Faça um programa que leia várias palavras pelo teclado, e armazene cada palavra em uma string. Depois, concatene todas as strings lidas numa única string. Por fim apresente esta como resultado ao final do programa.

5.3. Matrizes

5.3.1. Matrizes bidimensionais

Já vimos como declarar matrizes unidimensionais (vetores). Vamos tratar agora de matrizes bidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

tipo_da_variável nome_da_variável [altura][largura];

É muito importante ressaltar que, nesta estrutura, o índice da esquerda indexa as linhas e o da direita indexa as colunas. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador. Abaixo damos um exemplo do uso de uma matriz:

```
#include <stdio.h>
main ()
{
int mtrx [20][10];
int i,j,count;
count=1;
for (i=0;i<20;i++)
        for (j=0;j<10;j++)
                {
                mtrx[i][j]=count;
                count++;
                }
```

```
}
```

No exemplo acima, a matriz **mtrx** é preenchida, seqüencialmente por linhas, com os números de 1 a 200. Você deve entender o funcionamento do programa acima antes de prosseguir.

5.3.2. Matrizes de strings

Matrizes de strings são matrizes bidimensionais. Imagine uma string. Ela é um vetor. Se fizermos um vetor de strings estaremos fazendo uma lista de vetores. Esta estrutura é uma matriz bidimensional de chars. Podemos ver a forma geral de uma matriz de strings como sendo:

char nome_da_variável [num_de_strings][compr_das_strings];

Aí surge a pergunta: como acessar uma string individual? Fácil. É só usar apenas o primeiro índice. Então, para acessar uma determinada string faça:

nome_da_variável [índice]

Aqui está um exemplo de um programa que lê 5 strings e as exibe na tela:

```
#include <stdio.h>
main ()
{
char strings [5][100];
int count;
for (count=0;count<5;count++)
        {
        printf ("\n\nDigite uma string: ");
        gets (strings[count]);
        }
printf ("\n\n\nAs strings que voce digitou foram:\n\n");
for (count=0;count<5;count++)
        {
        printf ("%s\n",strings[count]);
        }
}
```

5.3.3. Matrizes multidimensionais

O uso de matrizes multidimensionais na linguagem C é simples. Sua forma geral é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

Uma matriz N-dimensional funciona basicamente como outros tipos de matrizes. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

5.3.4. Inicialização

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como inicialização é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN] = {lista_de_valores};

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 }
int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
char str [10] = "Joao";
char str_vect [3][10] = { "Joao", "Maria", "Jose" };
```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde **matrx** está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o ; no final da inicialização.

5.3.5. Inicialização sem especificação de tamanho

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho *a priori*. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```
char mess [] = "Linguagem C: flexibilidade e poder.";
int matrx [][2] = { 1,2,2,4,3,6,4,8,5,10 };
```

No primeiro exemplo, a string mess terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

5.3.6. AUTO AVALIAÇÃO

Veja como você está.
Faça um programa que leia 15 strings e as armazene numa matriz de strings. Depois de lidas as strings devem ser concatenadas numa só string (use um laço e verifique o tamanho das strings) e só então devem ser apresentadas na tela.

6. AULA 5 - PONTEIROS

A primeira linguagem que aprendi foi o BASIC. Nem tinha ouvido falar de ponteiros. Anos depois aprendi o Pascal. Programei em Pascal pelo menos três anos sem ouvir falar neles. Tive então uma pequena introdução aos ponteiros mas, sinceramente, não via muita utilidade. Esqueci os ponteiros e fui aprender uma nova linguagem: o Assembly. Na linguagem Assembly usa-se ponteiros quase que o tempo todo. Eu os usava, mas não sabia que eram ponteiros. Um ano depois aprendi a linguagem C. Voltei a ver ponteiros e descobri que aquilo que eu usava tanto na linguagem Assembly eram os próprios.
O C, como o Assembly, e diferentemente do BASIC e do Pascal, é altamente dependente dos ponteiros. Para ser um bom programador em C é fundamental que se tenha um bom domínio deles. Por isto, recomendo ao leitor um carinho especial com esta parte do curso que trata deles. Ponteiros são tão importantes na linguagem C que você já os viu e nem percebeu, pois mesmo para se fazer um introdução básica à linguagem C precisa-se deles.
Uma advertência: o uso descuidado de ponteiros pode levar a sérios bugs.

6.1. Como Funcionam os Ponteiros

Os ints guardam inteiros. Os floats guardam números de ponto flutuante. Os chars guardam caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.
Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.
Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.
No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro int aponta para um inteiro, isto é, guarda o endereço de um inteiro.

6.2. Declarando e Utilizando Ponteiros

Para declarar um ponteiro temos a seguinte forma geral:

*tipo_do_ponteiro *nome_da_variável;*

É o asterisco (*****) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp,*pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior. *O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!* Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e relocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count=10;
int *pt;
pt=&count;
```

Criamos um inteiro **count** com o valor 10 e um apontador para um inteiro **pt**. A expressão **&count** nos dá o endereço de count, o qual armazenamos em **pt**. Simples, não é? Repare que *não* alteramos o valor de **count**, que continua valendo 10.

Como nós colocamos um endereço em **pt**, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de **count** usando **pt**. Para tanto vamos usar o operador "inverso" do operador **&**. É o operador *****. No exemplo acima, uma vez que fizemos **pt=&count** a expressão ***pt** é equivalente ao próprio **count**. Isto significa que, se quisermos mudar o valor de count para 12, basta fazer ***pt=12**.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador ***** ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador ***** (multiplicação) não é o mesmo operador que o ***** (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário pré-fixado.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
main ()
{
int num,valor;
int *p;
num=55;
p=&num;     /* Pega o endereco de num */
valor=*p;       /* Valor e igualado a num de uma maneira
indireta */
printf ("\n\n%d\n",valor);
printf ("Endereco para onde o ponteiro aponta: %p\n",p);
printf ("Valor da variavel apontada: %d\n",*p);
```

```
}

#include <stdio.h>
main ()
{
int num,*p;
num=55;
p=&num;     /* Pega o endereco de num */
printf ("\nValor inicial: %d\n",num);
*p=100; /* Muda o valor de num de uma maneira indireta */
printf ("\nValor final: %d\n",num);
}
```

Nos exemplos acima vemos um primeiro exemplo do funcionamento dos ponteiros. No primeiro exemplo, o código **%p** usado na função **printf()** indica à função que ela deve exibir um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se temos dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1=p2**. Repare que estamos fazendo com que **p1** aponte para o mesmo lugar que **p2**. Se quisermos que a variável apontada por **p1** tenha o mesmo conteúdo da variável apontada por **p2** devemos fazer ***p1=*p2**. Basicamente, depois que se aprende a usar os dois operadores (**&** e *****) fica fácil entender operações com ponteiros.

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **int*** ele anda 2 bytes na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que **p** é um ponteiro, as operações são escritas como:

```
p++;
p--;
```

Mais uma vez insisto. Estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira. Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (== e **!=**). No caso de operações do tipo >, <, >= e <= estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*. Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
p1>p2
```

Há entretanto operações que você *não* pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair floats ou doubles de ponteiros.

6.2.1. AUTO AVALIAÇÃO

Veja como você está. Explique a diferença entre

p++;   (*p)++;   *(p++);

O que quer dizer *(p+10);? Explique o que você entendeu da comparação entre ponteiros

6.3. Ponteiros e Matrizes

Veremos nestas seções que ponteiros e matrizes têm uma ligação muito forte.

6.3.1. Matrizes como ponteiros

Vamos dar agora uma idéia de como o C trata matrizes.
Quando você declara uma matriz da seguinte forma:
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];
o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:
tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo
O compilador então aloca este número de bytes em um espaço livre de memória. O *nome da variável* que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*. Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz.
Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?
nome_da_variável[índice]
Isto pode ser facilmente explicado desde que você entenda que a notação acima é *absolutamente equivalente* a se fazer:
**(nome_da_variável+índice)*
Agora podemos entender como é que funciona uma matriz! Vamos ver o que podemos tirar de informação deste fato. Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de uma matriz, queremos, de fato, ***nome_da_variável** e então devemos ter um índice igual a zero. Então sabemos que:
**nome_da_variável* é equivalente a *nome_da_variável[0]*
Outra coisa: apesar de, na maioria dos casos, não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes da matriz. Isto explica também porque o C não verifica a validade dos índices. Ele *não* sabe o tamanho da matriz. Ele apenas aloca a memória, ajusta o ponteiro do nome da matriz para o início da mesma e, quando você usa os índices, encontra os elementos requisitados.
Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura seqüencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma seqüencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
main ()
{
float matrx [50][50];
int i,j;
for (i=0;i<50;i++)
        for (j=0;j<50;j++)
                matrx[i][j]=0.0;
}
```

Podemos reescrevê-lo de uma maneira muito mais eficiente como sendo:

```
main ()
{
float matrx [50][50];
float *p;
int count;
p=matrx[0];
for (count=0;count<2500;count++)
        {
        *p=0.0;
        p++;
        }
}
```

Você consegue ver porque é que o segundo programa é mais eficiente? Simplesmente porque *cada* vez que se faz **matrx[i][j]** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.
Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Seja:

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;

/* as operacoes a seguir sao invalidas */

vetor = vetor + 2;      /* ERRADO: vetor nao e' variavel */
vetor++;                /* ERRADO: vetor nao e' variavel */
vetor = ponteiro;       /* ERRADO: vetor nao e' variavel */
```

Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que vetor não é um Lvalue. Lvalue, significa "Left value", um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, isto é, uma variável. Outros dirão que tem-se "incompatible types in assignment", tipos incompatíveis em uma atribuição.

```
/* as operacoes abaixo sao validas */

ponteiro = vetor;       /* CERTO: ponteiro e' variavel */
ponteiro = vetor+2;     /* CERTO: ponteiro e' variavel */
```

O que você aprendeu nesta seção é de suma importância. Não siga adiante antes de entendê-la bem.

6.3.2. Ponteiros como matrizes

Sabemos agora que, na verdade, o nome de uma matriz é um ponteiro constante. Sabemos também que podemos indexar o nome de uma matriz. Como conseqüência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>
main ()
{
int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *p;
p=matrx;
printf ("O terceiro elemento da matriz e: %d",p[2]);
}
```

Podemos ver que **p[2]** eqüivale a ***(p+2)**.

6.3.3. Strings

Seguindo o raciocínio acima, nomes de strings, são do tipo **char***. Isto nos permite escrever a nossa função **StrCpy()**, que funcionará como a função strcpy() da biblioteca:

```
#include <stdio.h>
#include <string.h>
StrCpy (char *destino,char *origem)
{
while (*origem)
        {
        *destino=*origem;
        origem++;
        destino++;
        }
```

```
*destino='\0';
}
main ()
{
char str1[100],str2[100],str3[100];
printf ("Entre com uma string: ");
gets (str1);
StrCpy (str2,str1);
StrCpy (str3,"Voce digitou a string ");
printf ("\n\n%s%s",str3,str2);
}
```

Há vários pontos a destacar no programa acima. Observe que podemos passar ponteiros como argumentos de funções. Na verdade é assim que funções como **gets()** e **strcpy()** funcionam. Passando o ponteiro você possibilita à função *alterar* o conteúdo das strings. Você já estava passando os ponteiros e não sabia. No comando **while (*origem)** estamos usando o fato de que a string termina com '\0' como critério de parada. Quando fazemos **origem++** e **destino++** o leitor poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que recomendei que se deveria fazer, no final de uma seção anterior. O que o leitor talvez não saiba ainda (e que será estudado em detalhe mais adiante) é que, no C, são passados para as funções *cópias* dos argumentos. Desta maneira, quando alteramos o ponteiro **origem** na função **StrCpy()** o ponteiro **str2** permanece inalterado na função **main()**.

6.3.4. Endereços de elementos de matrizes

Nesta seção vamos apenas ressaltar que a notação

&nome_da_variável[índice]

é válida e retorna o endereço do ponto da matriz indexado por índice. É interessante notar que, como conseqüência, o ponteiro **nome_da_variável** tem o endereço **&nome_da_variável[0]**, que indica onde na memória está guardado o valor do primeiro elemento da matriz.

6.3.5. Matrizes de ponteiros

Podemos fazer matrizes de ponteiros como fazemos matrizes de qualquer outro tipo. Uma declaração de uma matriz de ponteiros inteiros poderia ser:

```
int *pmatrx [10];
```

No caso acima, **pmatrx** é um vetor que armazena 10 ponteiros para inteiros. Como existe uma certa equivalência entre vetores e ponteiros, podemos dizer que **pmatrx** é um ponteiro para ponteiro.

6.3.6. AUTO AVALIAÇÃO

Veja como você está.
Fizemos a função StrCpy(). Faça uma função StrLen() e StrCat() que funcionem como as funções strlen() e strcat() de string.h respectivamente

6.4. Inicializando Ponteiros

Podemos inicializar ponteiros. Vamos ver um caso interessante dessa inicialização de ponteiros com strings.
Precisamos para isto entender como o C trata as strings constantes. Toda string que o programador insere no programa é colocada num banco de strings que o compilador cria. No local onde está uma string no programa, o compilador coloca o endereço do início daquela string (que está no banco de strings). É por isto que podemos usar strcpy() do seguinte modo:

```
strcpy (string,"String constante.");
```

strcpy() pede dois parâmetros do tipo **char***. Como o compilador substitui a string **"String constante."** pelo seu endereço no banco de strings, tudo está bem para a função strcpy().

O que isto tem a ver com a inicialização de ponteiros? É que, para uma string que vamos usar várias vezes, podemos fazer:

```
char *str1="String constante.";
```

Aí poderíamos, em todo lugar que precisarmos da string, usar a variável **str1**. Devemos apenas tomar cuidado ao usar este ponteiro. Se o alterarmos vamos perder a string. Se o usarmos para alterar a string podemos facilmente corromper o banco de strings que o compilador criou.
Mais uma vez fica o aviso: ponteiros são poderosos mas, se usados com descuido, podem ser uma ótima fonte de dores de cabeça.

6.4.1. AUTO AVALIAÇÃO

Veja como você está. Escreva a função

```
int strend(char *s, char *t)
```

que retorna 1 (um) se a cadeia de caracteres 't' ocorrer no final da cadeia 's', e 0 (zero) caso contrário.

6.5. Ponteiros para Ponteiros

Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

*tipo_da_variável **nome_da_variável;*

Algumas considerações: ****nome_da_variável** é o conteúdo final da variável apontada; ***nome_da_variável** é o conteúdo do ponteiro intermediário.
No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros (UFA!) e assim por diante. Para fazer isto (não me pergunte a utilidade disto!) basta aumentar o número de asteriscos na declaração. A lógica é a mesma.

6.6. Cuidados a Serem Tomados ao se Usar Ponteiros

O principal cuidado ao se usar um ponteiro deve ser: saiba sempre *para onde* o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como *não* usar um ponteiro:

```
main () /* Errado - Nao Execute */
{
int x,*p;
x=13;
*p=x;
}
```

Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro p pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

6.6.1. AUTO AVALIAÇÃO

Veja como você está.
Escreva um programa que declare uma matriz 100x100 de inteiros. Você deve inicializar a matriz com zeros usando ponteiros. Preencha depois a matriz com os números de 1 a 10000 usando ponteiros.

7. AULA FUNÇÕES

7.1. A Função

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos temos de construi-lo bloco a bloco.
Uma função no C tem a seguinte forma geral:

tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
corpo_da_função
}

O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo int, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:

tipo nome1,tipo nome2, ... ,tipo nomeN

Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no tipo-de-retorno).
O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

7.2. O Comando return

O comando **return** tem a seguinte forma geral:

return valor_de_retorno; ou *return;*

Digamos que uma função está sendo executada. Quando se chega a uma declaração **return** a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser, pelo menos, compatível com o tipo de retorno declarado para a função.
Uma função pode ter mais de uma declaração **return**. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração **return**. Abaixo estão dois exemplos de uso do **return**:

```
#include <stdio.h>
int Square (int a)
{
return (a*a);
}
main ()
{
int num;
printf ("Entre com um numero: ");
scanf ("%d",&num);
num=Square(num);
printf ("\n\nO seu quadrado vale: %d\n",num);
}

#include <stdio.h>
int EPar (int a)
{
if (a%2)                /* Verifica se a e divisivel por dois
*/
        return 0;
else
        return 1;
}
main ()
{
```

```
int num;
printf ("Entre com um numero: ");
scanf ("%d",&num);
if (EPar(num))
        printf ("\n\nO numero e par.\n");
else
        printf ("\n\nO numero e impar.\n");
}
```

É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições. Mas *não* podemos fazer:

```
func(a,b)=x;    /* Errado! */
```

No segundo exemplo vemos o uso de mais de um **return** em uma função.
Fato importante: se uma função retorna um valor você *não precisa aproveitar* este valor. Se você não fizer nada com o valor de retorno de uma função ele será descartado. Por exemplo, a função **printf()** retorna um inteiro que nós nunca usamos para nada. Ele é descartado.

7.2.1. AUTO AVALIAÇÃO

Veja como você está. Escreva a função 'EDivisivel(int a, int b)' (tome como base EPar(int a)). A função deverá retornar 1 se o resto da divisão de a por b for zero. Caso contrário, a função deverá retornar zero.

7.3. Protótipos de Funções

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função **main()**. Isto é, as funções estão fisicamente antes da função **main()**. Eu fiz assim por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto que pus as funções antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos.
Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos usando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?
A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:

```
#include <stdio.h>
float Square (float a);
main ()
{
float num;
printf ("Entre com um numero: ");
scanf ("%f",&num);
num=Square(num);
printf ("\n\nO seu quadrado vale: %f\n",num);
}
float Square (float a)
{
return (a*a);
}
```

Observe que a função **Square()** está colocada depois de **main()**, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.
Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Eles não só ajudam o compilador. Eles ajudam a você também. Usando protótipos o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que já é uma grande ajuda.

7.4. O Tipo void

Agora vamos ver o único tipo da linguagem C que não vimos ainda: o **void**. Em inglês, **void** quer dizer vazio e é isto mesmo que o **void** é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros! Podemos agora escrever o protótipo de uma função que não retorna nada:

void nome_da_função (declaração_de_parâmetros);

Numa função, como a acima, não temos valor de retorno na declaração return. Aliás, neste caso, o comando return não é necessário na função.
Podemos, também, fazer funções que não têm parâmetros:

tipo_de_retorno nome_da_função (void);

ou, ainda, que não tem parâmetros e não retornam nada:

void nome_da_função (void);

Um exemplo de funções que usam o tipo **void**:

```
#include <stdio.h>
void Mensagem (void);
main ()
{
Mensagem();
printf ("\tDiga de novo:\n");
Mensagem();
}
void Mensagem (void)
{
printf ("Ola! Eu estou vivo.\n");
}
```

Usando nossos conhecimentos de funções, podemos agora eliminar de vez aquela irritante mensagem de aviso que o compilador tem nos dado, muitas vezes, desde o início do nosso curso! Se quisermos que a função retorne algo, devemos usar a declaração return. Se não quisermos, basta declarar a função como tendo tipo-de-retorno **void**. Devemos lembrar agora que a função **main()** *é* uma função e como tal devemos tratá-la. O compilador acha que a função **main()** deve retornar um inteiro. Isto pode ser interessante se quisermos que o sistema operacional receba um valor de retorno da função **main()**. Se assim o quisermos, devemos nos lembrar da seguinte convenção: se o programa retornar zero, significa que ele terminou normalmente, e, se o programa retornar um valor diferente de zero, significa que o programa teve um termino anormal. Se não estivermos interessados neste tipo de coisa, basta declarar a função main como retornando **void**.
As duas funções **main()** abaixo são válidas e não geram mensagens de aviso:

```
main (void)
{
....
return 0;
}

void main (void)
{
....
}
```

Pronto. Estamos livres daquelas mensagens de aviso.

7.5. Arquivos-Cabeçalhos

Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em **.h**. Já vimos exemplos como **stdio.h**, **conio.h**, **string.h**. Eles, na verdade, não possuem os códigos completos das funções. Eles só contêm *protótipos* de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem". Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.

Se você programar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-cabeçalhos e incluí-los também.

Suponha que a função 'int EPar(int a)', do segundo exemplo das páginas anterior seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado por exemplo de 'funcao.h' teremos a seguinte declaração:

```
int EPar(int a);
```

O código da função será escrito num arquivo a parte. Vamos chamá-lo de 'funcao.c'. Neste arquivo teremos a definição da função:

```
int EPar (int a)
{
if (a%2)                /* Verifica se a e divisivel por dois
*/
        return 0;
else
        return 1;
}
```

Por fim, no arquivo do programa principal teremos o programa principal. Vamos chamar este arquivo aqui de 'princip.c'.

```
#include <STDIO.H>
#include "funcao.h"
main ()
{
int num;
printf ("Entre com numero: ");
scanf ("%d",&num);
if (EPar(num))
        printf ("\n\nO numero e par.\n");
else
        printf ("\n\nO numero e impar.\n");
}
```

Este programa poderia ser compilado usando a seguinte linha de comando para o gcc:

```
gcc princip.c funcao.c -o saida
```

onde 'saida' seria o arquivo executável gerado.

7.6. Escopo de Variáveis

Já foi dada uma introdução ao escopo de variáveis. O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa.

7.6.1. Variáveis locais

O primeiro tipo de variáveis que veremos são as variáveis locais. Estas são aquelas que só têm validade dentro do bloco no qual são declaradas. Sim. Podemos declarar variáveis dentro de qualquer bloco. Só para lembrar: um bloco começa quando abrimos uma chave e termina quando fechamos a chave. Até agora só tínhamos visto variáveis locais para funções completas.

Mas um comando for pode ter variáveis locais e que não serão conhecidas fora dali. A declaração de variáveis locais é a primeira coisa que devemos colocar num bloco. A característica que torna as variáveis locais tão importantes é justamente a de serem exclusivas do bloco. Podemos ter quantos blocos quisermos com uma variável local chamada **x**, por exemplo, e elas não apresentarão conflito entre elas.
A palavra reservada do C **auto** serve para dizer que uma variável é local. Mas não precisaremos usá-la pois as variáveis declaradas dentro de um bloco já são consideradas locais.
Abaixo vemos um exemplo de variáveis locais:

```
func1 (...)
{
int abc,x;
...
}
func (...)
{
int abc;
...
}
main ()
{
int a,x,y;
for (...)
        {
        float a,b,c;
        ...
        }
...
}
```

No programa acima temos três funções. As variáveis locais de cada uma delas não irão interferir com as variáveis locais de outras funções. Assim, a variável **abc** de **func1()** não tem nada a ver (e pode ser tratada independentemente) com a variável **abc** de **func2()**. A variável **x** de **func1()** é também completamente independente da variável **x** de **main()**. As variáveis **a**, **b** e **c** são locais ao bloco for. Isto quer dizer que só são conhecidas dentro deste bloco for e são desconhecidas no resto da função **main()**. Quando usarmos a variável **a** dentro do bloco for estaremos usando a variável a local ao for e não a variável **a** da função **main()**.

7.6.2. Parâmetros formais

O segundo tipo de variável que veremos são os parâmetros formais. Estes são declarados como sendo as entradas de uma função. Não há motivo para se preocupar com o escopo deles. É fácil: o parâmetro formal é como se fosse uma variável local da função. Você pode também alterar o valor de um parâmetro formal, pois esta alteração não terá efeito na variável que foi passada à função. Isto tem sentido, pois quando o C passa parâmetros para uma função, são passadas apenas cópias das variáveis. Isto é, os parâmetros formais existem independentemente das variáveis que foram passadas para a função. Eles tomam apenas uma cópia dos valores passados para a função.

7.6.3. Variáveis globais

Variáveis globais são declaradas, como já sabemos, fora de todas as funções do programa. Elas são conhecidas e podem ser alteradas por todas as funções do programa. Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local. Vamos ver um exemplo:

```
int z,k;
func1 (...)
{
int x,y;
...
```

```
}
func2 (...)
{
int x,y,z;
...
z=10;
...
}
main ()
{
int count;
...
}
```

No exemplo acima as variáveis **z** e **k** são globais. Veja que **func2()** tem uma variável local chamada **z**. Quando temos então, em **func2()**, o comando **z=10** quem recebe o valor de 10 é a variável *local*, não afetando o valor da variável global **z**.
Evite *ao máximo* o uso de variáveis globais. Elas ocupam memória o tempo todo (as locais só ocupam memória enquanto estão sendo usadas) e tornam o programa mais difícil de ser entendido e menos geral.

7.6.4. AUTO AVALIAÇÃO

Veja como você está. Estude o seguinte programa e aponte o valor de cada variável sempre que solicitado:

```
#include <stdio.h>
int num;
int func(int a, int b)
{
a = (a+b)/2;  /* Qual e o valor de a apos a atribuicao? */
num -= a;
return a;
}

main()
{
int first = 0, sec = 50;
num = 10;
num += func(first, sec);  /* Qual e o valor de num, first e sec */
                          /* antes e depois da atribuicao?      */
printf("\n\nConfira! num = %d\tfirst = %d\tsec = %d",num, first,
sec);
}
```

7.7. *Chamada por Valor e Chamada por Referência*

Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é chamado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios. Veja o exemplo abaixo:

```
#include <stdio.h>
float sqr (float num);
void main (void)
{
float num,sq;
printf ("Entre com um numero: ");
scanf ("%f",&num);
sq=sqr(num);
printf ("\n\nO numero original e: %f\n",num);
```

```
printf ("O seu quadrado vale: %f\n",sq);
}
float sqr (float num)
{
num=num*num;
return num;
}
```

No exemplo acima o parâmetro formal **num** da função **sqr()** sofre alterações dentro da função, mas a variável **num** da função **main()** permanece inalterada: é uma chamada por valor.
Outro tipo de chamada de função é quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).
O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar mexendo nos valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recurso que permite ao programador fazer chamadas por referência. Há entretanto, no C, um recurso de programação que podemos usar para simular uma chamada por referência.
Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um **&** na frente das variáveis que estivermos passando para a função. Veja um exemplo:

```
#include <stdio.h>
void Swap (int *a,int *b);
void main (void)
{
int num1,num2;
num1=100;
num2=200;
Swap (&num1,&num2);
printf ("\n\nEles agora valem %d  %d\n",num1,num2);
}
void Swap (int *a,int *b)
{
int temp;
temp=*a;
*a=*b;
*b=temp;
}
```

Não é muito difícil. Espere um momento... será que nós já não vimos esta história de chamar uma função com as variáveis precedidas de **&**? Já. É assim que nós chamamos a função **scanf()**. Mas porquê? Vamos pensar um pouco. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e portanto precisa alterar seus valores.

7.7.1. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia uma matriz de inteiros pelo teclado e a apresente na tela. Use uma função (lematriz(int *mat)) para fazer a leitura da matriz.

7.8. *Matrizes como Argumentos de Funções*

Quando vamos passar uma matriz como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Digamos que temos a seguinte matriz:

```
int matrx [50];
```

e que queiramos passá-la como argumento de uma função **func()**. Podemos declarar **func()** das três maneiras seguintes:

```
void func (int matrx[50]);
void func (int matrx[]);
void func (int *matrx);
```

Veja que, nos três casos, teremos dentro de **func()** um **int*** chamado **matrx**. Note que, no caso de estarmos passando uma matriz para uma função, teremos de passá-la através de um ponteiro. Isto faz com que possamos alterar o valor desta matriz dentro da função. Todo cuidado é pouco! Um exemplo disto já foi visto quando implementamos a função **StrCpy()**.

7.9. Os Argumentos argc e argv

A função **main()** pode ter parâmetros formais. Mas o programador não pode escolher quais serão eles. A declaração mais completa que se pode ter para a função **main()** é:

```
int main (int argc,char *argv[]);
```

Os parâmetros **argc** e **argv** dão ao programador acesso à linha de comando com a qual o programa foi chamado.
O **argc** (argument count) é um inteiro e possui o número de argumentos com os quais a função **main()** foi chamada na linha de comando. Ele é, no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.
O **argv** (argument values) é um ponteiro para uma matriz de strings. Cada string desta matriz é um dos parâmetros da linha de comando. O **argv[0]** sempre aponta para o nome do programa (que, como já foi dito, é considerado o primeiro argumento). É para saber quantos elementos temos em **argv** que temos **argc**.

7.9.1. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que faça uso dos parâmetros *argv* e *argc*. O programa deverá receber da linha de comando o dia, mês e ano correntes, e imprimir a data em formato apropriado. Veja o exemplo, supondo que o executável se chame data:

```
data 26 04 1998
```

O programa deverá imprimir um dos seguintes resultados:

```
26/04/1998
26/Abril/1998
26 de abril de 1998
```

7.10. Recursividade

Na linguagem C, assim como em muitas outras linguagens de programação, uma função pode chamar a si própria. Uma função assim é chamada função recursiva. Todo cuidado é pouco ao se fazer funções recursivas. A primeira coisa a se providenciar é um critério de parada. Este vai determinar quando a função deverá parar de chamar a si mesma. Isto impede que a função se chame infinitas vezes.
A função fatorial(int n) pode ser um bom exemplo de uma função recursiva:

```
#include <stdio.h>
int fat(int n)
{
int ret;
if (n)
  return n*fat(n-1)
else return 1;
}

void main()
{
int n;
printf("\n\nDigite um valor para n: ");
```

```
        scanf("%d", &n);
        printf("\nO fatorial de %d e' %d", n, fat(n));
        }
```

Há certos algoritmos que são mais eficientes quando feitos de maneira recursiva, mas a recursividade é algo a ser evitado sempre que possível, pois, se usada incorretamente, tende a consumir muita memória e ser lenta. Lembre-se que memória é consumida cada vez que o computador faz uma chamada a uma função. Com funções recursivas a memória do computador pode se esgotar rapidamente.

7.11. Outras Questões

Uma função, como foi dito anteriormente, é um bloco de construção muito útil. No C as funções são flexíveis. A flexibilidade dá poder, mas exige cuidado.
Funções devem ser implementadas, quando possível, da maneira mais geral possível. Isto as torna mais fáceis de serem reutilizadas e entendidas. Evite, sempre que possível, funções que usem variáveis globais.
Se houver uma rotina que deve ser o mais veloz possível, seria bom implementá-la sem nenhuma (ou com o mínimo de) chamada a funções, porque uma chamada a uma função consome tempo e memória.
Um outro ponto importante é que, como já sabemos um bocado a respeito de funções, quando formos ensinar uma das funções das bibliotecas do C vamos mostrar, em primeiro lugar, o seu protótipo. Quem entendeu tudo que foi ensinado nesta parte sobre funções pode retirar inúmeras informações de um protótipo (tipo de retorno, nome da função, tipo dos argumentos, passagem por valor ou passagem por referência).
Sugiro, que neste ponto, o leitor leia um arquivo-cabeçalho como, por exemplo o **conio.h** ou o **string.h**. É um bom treino. Estes arquivo podem ser encontrados no diretório apropriado do compilador que você estiver utilizando. Consulte o manual do compilador.

7.11.1. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia uma frase do teclado e imprima o número de caracteres total, o número de palavras e o número médio de letras por palavra para a frase. Evite usar as funções das bibliotecas neste ponto. É importante reaproveitar mas este momento é o momento que vocês estão aprendendo.

7.12. Ponteiros para Funções

Até agora usamos ponteiros para apontar para endereços de memória onde se encontravam as variáveis (dados). Algumas vezes é necessário apontar para funções, isto é, apontar para o endereço de memória que contem o inicio das instruções de uma função. Quando assim procedemos, dizemos que usaremos *ponteiros para funções*.

7.12.1. Ponteiros como chamada de função.

Um uso de ponteiros para funções é passar uma função como **argumento** de outra função. Mas também se pode usar ponteiros para funções ao invés de funções nas chamadas normais de funções.
Sintaxe: A sintaxe de declaração de ponteiro para funções é a seguinte:

*tipo_r (*nome_p)(lista);*

onde:

tipo_r é o tipo de retorno da função apontada.
nome_p é o nome do ponteiro que apontara para a função.
lista é a lista de argumentos da função.

Exemplo: Suponha que temos uma função é declarada como:

```
float fun(int a, int b){
    ...
    }
```

o ponteiro correspondente será:

```
float (*pt)(int,int);
```

Observe que o ponteiro para função **deve ser** declarado entre parênteses. Observe também que o ponteiro e a função retornam o mesmo tipo de dado e que tem os mesmos argumentos.

Sintaxe: Para atribuirmos o endereço de uma função para um ponteiro usamos a seguinte sintaxe:

```
pont = &função;
```

onde:

pont é o nome do ponteiro.
função é o nome da função.

Se um ponteiro contem o endereço de uma função, ele pode ser usado no lugar da chamada da função.

Exemplo: o trecho de programa abaixo usa um ponteiro para chamar uma função:

```
float fun(int a,int b){
  ...
}
void main(void){
  float temp;
  float (*pt)(int,int);
  pt = &fun;
  temp = (*pt)(10,20); // eqüivale a: temp = fun(10,20);
  ...
}
```

7.12.2.Passando uma função como argumento de outra função.

Outra utilização de ponteiros para funções é na passagem de uma **função** como **argumento** para outra função. Para que isso ocorra necessitamos:

Na declaração da **função a ser passada**:

i) Nada de especial, apenas a definição normal:

```
tipo nome_p(lista){
  ...
}
```

Exemplo:

```
float soma(float a,float b){
  ...
}
```

*1Na **função receptora**:

i) Declarar o **ponteiro** que recebe a **função passada** na lista de argumentos:

```
tipo nome_r(..., tipo (*pt)(lista), ...){
```

Exemplo:

```
float grad(float x, float y, float (*p)(float,float)){
```

ii) Usar o ponteiro para funções nas chamadas da função passada:

```
var = (*pt)(lista);
```

Exemplo:

```
valor = (*p)(x,y);
```

*2Na função principal:

i) Passar o nome da função chamada para a função receptora:

```
var = nome_g(... , nome_p , ...);
```

Exemplo:

```
g = grad(x,y,soma);
```

8. AULA 7 - DIRETIVAS DE COMPILAÇÃO

8.1. As Diretivas de Compilação

Diretivas de compilação são comandos que não são compilados. Eles são dirigidos ao compilador. Já vimos, e usamos muito, a diretiva #include. Sabemos que ela não gera código mas diz ao compilador que ele deve incluir um arquivo externo na hora da compilação. As diretivas do C são identificadas por começarem por #. As diretivas definidas pelo padrão ANSI são:

Diretivas
#if
#else
#include
#line
#ifdef
#elif
#define
#error
#ifndef
#endif
#undef
#pragma

Como o conjunto de diretivas varia de um compilador para outro, procuraremos ser breves em suas descrições. Para maiores informações aconselhamos o leitor a procurar o manual específico do compilador que está sendo utilizado.

8.2. A Diretiva include

A diretiva **#include** já foi usada durante o nosso curso diversas vezes. Ela diz ao compilador para incluir, na hora da compilação, um arquivo especificado. Sua forma geral é:

#include "nome_do_arquivo" ou

#include <nome_do_arquivo>

A diferença entre se usar **""** e **<>** é somente a ordem de procura nos diretórios pelo arquivo especificado. Se você quiser informar o nome do arquivo com o caminho completo, ou se o arquivo estiver no diretório de trabalho, use **""**. Se o arquivo estiver nos caminhos de procura pré-especificados do compilador (como é o caso de arquivos como **stdio.h**) use **<>**.

8.3. As Diretivas define e undef

A diretiva **#define** tem a seguinte forma geral:

#define nome_da_macro sequência_de_caracteres

Quando você usa esta diretiva, você está dizendo ao compilador para que, toda vez que ele encontrar o nome-da-macro no programa a ser compilado, ele deve substituí-lo pela seqüência de caracteres fornecida. Isto é muito útil para deixar o programa mais geral. Veja um exemplo:

```
#include <stdio.h>
#define PI 3.1416
#define VERSAO "2.02"
main ()
{
printf ("Programa versao %s",VERSAO);
printf ("O numero pi vale: %f",PI);
}
```

Se quisermos mudar o nosso valor de **PI**, ou da **VERSAO**, no programa acima, basta mexer no início do programa. Isto torna o programa mais flexível. Há quem diga que, em um programa, nunca se deve usar constantes como 10, 3.1416, etc., pois estes são números que ninguém sabe o que significam. Ao invés disto deve-se usar apenas **#define**s. É uma convenção na linguagem C que as macros declaradas em **#define**s devem ser todas em maiúsculas. Esta é uma convenção que deve ser seguida pois torna o programa mais legível.

Um outro uso da diretiva #define é o de simplesmente definir uma macro. Neste caso usa-se a seguinte forma geral:

#define nome_da_macro

Neste caso o objetivo não é usar a macro no programa (pois ela seria substituída por nada), mas, sim, definir uma macro para ser usada como uma espécie de flag. Isto quer dizer que estamos definindo um valor como sendo "verdadeiro" para depois podermos testá-lo.

Também é possível definir macros com argumentos. Veja o exemplo a seguir:

```
#define max(A,B) ((A>B) ? (A):(B))
#define min(A,B) ((A<B) ? (A):(B))
...
x = max(i,j);
y = min(t,r);
```

Embora pareça uma chamada de função, o uso de `max` (ou `min`) simplesmente substitui, em tempo de compilação, o código especificado. Cada ocorrência de um parâmetro formal (A ou B, na definição) será substituído pelo argumento real correspondente. Assim, a linha de código

```
x = max(p+q,r+s);
```

será substituída pela linha:

```
x = ((p+q)>(r+s) ? (p+q):(r+s));
```

Isto pode ser muito útil. Verifique que as macros `max` e `min` não possuem especificação de tipo. Logo, elas trabalham corretamente para qualquer tipo de dado, enquanto os argumentos passados forem coerentes. Mas isto pode trazer também algumas armadilhas. Veja que a linha

```
x = max(p++,r++);
```

será substituída pelo código

```
x = ((p++)>(r++) ? (p++):(r++));
```

e em conseqüência, incrementará o maior valor duas vezes.

A diretiva **#undef** tem a seguinte forma geral:

#undef nome_da_macro

Ela faz com que a macro que a segue seja apagada da tabela interna do compilador que guarda as macros. É como se o compilador não conhecesse mais esta macro.

8.4. As Diretivas ifdef e endif

Nesta seção, e até mais a frente, veremos as diretivas de compilação condicional. Elas são muito parecidas com os comandos de execução condicional do C. As duas primeiras diretivas que veremos são as **#ifdef** e **#endif**. Suas formas gerais são:

#ifdef nome_da_macro
sequência_de_declarações
#endif

A seqüência de declarações será compilada apenas se o nome da macro estiver definido. A diretiva de compilação **#endif** é útil para definir o fim de uma seqüência de declarações para todas as diretivas de compilação condicional. As linhas

```
#define PORT0 0x378
...
/* Linhas de codigo qualquer... */
...
#ifdef PORT_0
  #define PORTA PORT_0
  #include "../sys/port.h"
#endif
```

demonstram como estas diretivas podem ser utilizadas. Caso *PORT_0* tenha sido previamente definido, a macro *PORTA* é definida e o header file *port.h* é incluído.

8.5. A Diretiva ifndef

A diretiva **#ifndef** funciona ao contrário da diretiva #ifdef. Sua forma geral é:

#ifndef nome_da_macro

sequência_de_declarações

#endif

A seqüência de declarações será compilada se o nome da macro não tiver sido definido.

8.6. A Diretiva if

A diretiva **#if** tem a seguinte forma geral:

#if expressão_constante

sequência_de_declarações

#endif

A seqüência de declarações será compilada se a expressão-constante for verdadeira. É muito importante ressaltar que a expressão fornecida deve ser constante, ou seja, não deve ter nenhuma variável.

8.7. A Diretiva else

A diretiva **#else** tem a seguinte forma geral:

#if expressão_constante

sequência_de_declarações

#else

sequência_de_declarações

#endif

Ela funciona como seu correspondente, o comando else.
Imagine que você esteja trabalhando em um sistema, e deseje que todo o código possa ser compilado em duas diferentes plataformas (i.e. Unix e Dos). Para obter isto, você "encapsula" toda a parte de entrada e saída em arquivos separados, que serão carregados de acordo com o header file carregado. Isto pode ser facilmente implementado da seguinte forma:

```
#define SISTEMA DOS
...
/*linhas de codigo..*/
...
#if SISTEMA == DOS
  #define CABECALHO "dos_io.h"
#else
  #define CABECALHO "unx_io.h"
#endif
#include CABECALHO
```

8.8. A Diretiva elif

A diretiva **#elif** serve para implementar a estrutura if-else-if. Sua forma geral é:

```
#if expressão_constante_1
   sequência_de_declarações_1
#elif expressão_constante_2
   sequência_de_declarações_2
#elif expressão_constante_3
   sequência_de_declarações_3
.
.
.
#elif expressão_constante_n
   sequência_de_declarações_n
#endif
```

O funcionamento desta estrutura é idêntico ao funcionamento apresentado anteriormente.

8.9. A Diretiva line

A diretiva **#line** é uma diretiva pouco usada. Sua finalidade é alterar os valores de duas macros já pré-definidas pelo compilador. São as macros **__LINE__** e **__FILE__**. Sua forma geral é:

#line número "nome_do_arquivo"

O número deve ser um inteiro positivo e fornece ao compilador o número de linha que ele estaria compilando no presente momento. O nome de arquivo é opcional e fornece ao compilador o nome do arquivo que ele estaria compilando atualmente. Note que você estaria "enganando" o compilador, fornecendo dados "falsos" a ele, pois ele mantém um registro correto do número de linha e nome de arquivo.

8.10. A Diretiva error

A diretiva **#error** gera um erro de compilação. Ela tem a seguinte forma geral:

#error mensagem_de_erro

Quando o compilador encontra a diretiva, ele pára a compilação e apresenta ao usuário a mensagem de erro.

8.11. A Diretiva pragma

A diretiva **#pragma** é definida pela implementação e permite que várias instruções sejam dadas ao compilador. Por exemplo, um compilador pode ter uma opção que suporte uma execução passo a passo (*tracing*) do programa. Uma opção de "trace" seria, então, especificada por um comando **#pragma**. Será necessário consultar o manual do compilador para obter detalhes e opções.

9. AULA 8 - Entradas e Saídas Padronizadas

9.1. Introdução

As entradas e saídas padronizadas do C fornecem ao programador uma interface consistente. Esta interface se presta a entradas e saídas na tela texto e a entradas e saídas em arquivos. Já vimos algumas das funções apresentadas aqui. Não vamos entrar em detalhes nestas seções pois estas funções podem ser encontradas no manual do seu compilador.
Não é objetivo deste curso explicar, em detalhes, funções. A sintaxe completa das funções do C podem ser encontradas no manual do seu compilador. Alguns compiladores trazem um descrição das funções na ajuda do compilador que pode ser acessada "on line".
Um ponto importante é que agora, quando apresentarmos uma função, vamos, em primeiro lugar, apresentar o seu protótipo. O leitor já deve ser capaz de interpretar as informações que um protótipo nos passa.

9.2. Lendo e Escrevendo Caracteres

Uma das funções mais básicas que um programador quer é a entrada e saída em dispositivos. Estes podem ser um monitor, uma impressora ou um arquivo em disco. Vamos ver os principais comandos que a padronização do C nos fornece para isto.

9.2.1. getche e getch

As funções getch() e getche() não são definidas pelo padrão ANSI. Porém, elas geralmente são incluídas em compiladores baseados no DOS, e se encontram no header file *conio.h*. Mas lembre-se: são funções comuns apenas para compiladores baseados em DOS.
Protótipos:

```
int getch (void);
int getche (void);
```

getch() espera que o usuário digite uma tecla e retorna este caractere. Você pode estar estranhando o fato de **getch()** retornar um inteiro, mas não há problema pois este inteiro é tal que quando igualado a um **char** a conversão é feita corretamente. A função **getche()** funciona exatamente como **getch()**. A diferença é que **getche()** gera um "echo" na tela antes de retornar a tecla.
Se a tecla pressionada for um caractere especial estas funções retornam zero. Neste caso você deve usar as funções novamente para pegar o código da tecla estendida pressionada.

9.2.2. putchar

Protótipo:

```
int putchar (int c);
```

putchar() coloca o caractere c na tela. Este caractere é colocado na posição atual do cursor. Mais uma vez os tipos são inteiros, mas você não precisa se preocupar com este fato. O header file é *stdio.h*.

9.3. Lendo e Escrevendo Strings

9.3.1. gets

Protótipo:

```
char *gets (char *s);
```

Pede ao usuário que entre uma string, que será armazenada na string **s**. O ponteiro que a função retorna é o próprio **s**.

9.3.2. puts

Protótipo:

```
int puts (char *s);
```

puts() coloca a string **s** na tela.

9.3.3. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia nomes pelo teclado e os imprima na tela. Use as funções puts e gets para a leitura e impressão na tela.

9.4. Entrada e Saída Formatada

As funções que resumem todas as funções de entrada e saída formatada no C são as funções **printf()** e **scanf()**. Um domínio destas funções é fundamental ao programador.

9.4.1. printf

Protótipo:

```
int printf (char *str,...);
```

As reticências no protótipo da função indicam que esta função tem um número de argumentos variável. Este número está diretamente relacionado com a string de controle **str**, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato. Como já vimos, os últimos determinam uma exibição de variáveis na saída. Os comandos de formato são precedidos de **%**. A cada comando de formato deve corresponder um argumento na função **printf()**. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.
Abaixo apresentamos a tabela de códigos de formato:

Código	Formato
%c	Um caracter (char)
%d	Um número inteiro decimal (int)
%i	O mesmo que %d
%e	Número em notação científica com o "e"minúsculo
%E	Número em notação científica com o "e"maiúsculo
%f	Ponto flutuante decimal
%g	Escolhe automaticamente o melhor entre %f e %e
%G	Escolhe automaticamente o melhor entre %f e %E
%o	Número octal
%s	String
%u	Decimal "unsigned" (sem sinal)
%x	Hexadecimal com letras minúsculas
%X	Hexadecimal com letras maiúsculas
%%	Imprime um %
%p	Ponteiro

Vamos ver alguns exemplos:

Código	Imprime
`printf ("Um %%%c %s",'c',"char");`	`Um %c char`
`printf ("%X %f %e",107,49.67,49.67);`	`6B 49.67 4.967e1`
`printf ("%d %o",10,10);`	`10 12`

É possível também indicar o tamanho do campo, justificação e o número de casas decimais. Para isto usa-se códigos colocados entre o **%** e a letra que indica o tipo de formato.

Um inteiro indica o tamanho mínimo, em caracteres, que deve ser reservado para a saída. Se colocarmos então **%5d** estamos indicando que o campo terá cinco caracteres de comprimento *no mínimo*. Se o inteiro precisar de mais de cinco caracteres para ser exibido então o campo terá o comprimento necessário para exibi-lo. Se o comprimento do inteiro for menor que cinco então o campo terá cinco de comprimento e será preenchido com espaços em branco. Se quiser um preenchimento com zeros pode-se colocar um zero antes do número. Temos então que **%05d** reservará cinco casas para o número e se este for menor então se fará o preenchimento com zeros.
O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se um sinal - antes do número de casas. Então **%-5d** será o nosso inteiro com o número mínimo de cinco casas, só que justificado a esquerda.
Pode-se indicar o número de casas decimais de um número de ponto flutuante. Por exemplo, a notação **%10.4f** indica um ponto flutuante de comprimento total dez e com 4 casas decimais. Entretanto, esta mesma notação, quando aplicada a tipos como inteiros e strings indica o número mínimo e máximo de casas. Então **%5.8d** é um inteiro com comprimento mínimo de cinco e máximo de oito.
Vamos ver alguns exemplos:

<table>
<tr><th>Código</th><th>Imprime</th></tr>
<tr><td><code>printf ("%-5.2f",456.671);</code></td><td><code>| 456.67|</code></td></tr>
<tr><td><code>printf ("%5.2f",2.671);</code></td><td><code>| 2.67|</code></td></tr>
<tr><td><code>printf ("%-10s","Ola");</code></td><td><code>|Ola |</code></td></tr>
</table>

Nos exemplos o "pipe" (|) indica o início e o fim do campo mas não são escritos na tela.

9.4.2. scanf

Protótipo:

```
int scanf (char *str,...);
```

A string de controle str determina, assim como com a função **printf()**, quantos parâmetros a função vai necessitar. Devemos sempre nos lembrar que a função **scanf()** deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador **&**. Os especificadores de formato de entrada são muito parecidos com os de **printf()**:

Código	Formato
%c	Um único caracter (char)
%d	Um número decimal (int)
%i	Um número inteiro
%e	Um ponto flutuante
%f	Um ponto flutuante
%h	Inteiro curto
%o	Número octal
%s	String
%x	Número hexadecimal
%p	Ponteiro

Os caracteres de conversão *d, i, u* e *x* podem ser precedidos por *h* para indicarem que um apontador para *short* ao invés de *int* aparece na lista de argumento, ou pela letra *l* (letra ele) para indicar que um apontador para *long* aparece na lista de argumento. Semelhantemente, os caracteres de conversão *e, f* e *g* podem ser precedidos por *l* para indicarem que um apontador para *double* ao invés de *float* está na lista de argumento.

9.4.3. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia (via teclado) e apresente uma matriz 3X3 na tela. Utilize os novos códigos de formato aprendidos para que a matriz se apresente corretamente indentada. Altere os tipos de dados da matriz (int, float, double) e verifique a

formatação correta para a ideação. Verifique também a leitura e impressão de números hexadecimais.

9.5. Abrindo e Fechando um Arquivo

No sistema de entrada e saída ANSI é definido o tipo "ponteiro de arquivo". Este não é um tipo propriamente dito, mas uma definição usando o comando typedef. Esta definição está no arquivo cabeçalho **stdio.h** ou **stdlib.h** dependendo do seu compilador. Podemos declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

p será então um ponteiro para um arquivo. É usando este tipo de ponteiro que vamos poder manipular arquivos no C.

9.5.1. fopen

Esta é a função de abertura de arquivos. Seu protótipo é:

```
FILE *fopen (char *nome_do_arquivo,char *modo);
```

O nome_do_arquivo determina qual arquivo deverá ser aberto. Este nome deve ser válido no sistema operacional que estiver sendo utilizado. O modo de abertura diz à função **fopen()** que tipo de uso você vai fazer do arquivo. A tabela abaixo mostra os valores de modo válidos:

Modo	Significado
"r"	Abre um arquivo para leitura
"w"	Cria um arquivo para escrita
"a"	Acrescenta dados no fim do arquivo ("append")
"rb"	Abre um arquivo binário para leitura
"wb"	Cria um arquivo binário para escrita
"ab"	Acrescenta dados binários no fim do arquivo
"r+"	Abre um arquivo para leitura e escrita
"w+"	Cria um arquivo para leitura e escrita
"a+"	Acrescenta dados ou cria uma arquivo para leitura e escrita
"r+b"	Abre um arquivo binário para leitura e escrita
"w+b"	Cria um arquivo binário para leitura e escrita
"a+b"	Acrescenta dados ou cria uma arquivo binário para leitura e escrita
"rt"	Abre um arquivo texto para leitura
"wt"	Cria um arquivo texto para escrita
"at"	Acrescenta dados no fim do arquivo texto
"r+t"	Abre um arquivo texto para leitura e escrita
"w+t"	Cria um arquivo texto para leitura e escrita
"a+t"	Acrescenta dados ou cria uma arquivo texto para leitura e escrita

Poderíamos então, para abrir um arquivo binário, escrever:

```
FILE *fp;
fp=fopen ("exemplo.bin","wb");
if (!fp)
        printf ("Erro na abertura do arquivo.");
```

A condição **!fp** testa se o arquivo foi aberto com sucesso porque no caso de um erro a função **fopen()** retorna um ponteiro nullo (**NULL**).

9.5.2. exit

Aqui abrimos um parênteses para explicar a função **exit()** cujo protótipo é:

```
void exit (int codigo_de_retorno);
```

Esta função aborta a execução do programa. Pode ser chamada de qualquer ponto no programa e faz com que o programa termine e retorne, para o sistema operacional, o código_de_retorno. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um número não nulo no caso de ter ocorrido um problema. A função **exit()** se torna importante em casos como alocação dinâmica e abertura de arquivos pois pode ser essencial que uma

determinada memória seja alocada ou que um arquivo seja aberto. Poderíamos rescrever o exemplo da seção anterior usando agora o exit() para garantir que o programa não deixará de abrir o arquivo:

```
#include <stdio.h
main (void)
{
FILE *fp;
...
fp=fopen ("exemplo.bin","wb");
if (!fp)
        {
        printf ("Erro na abertura do arquivo. Fim de
programa.");
        exit (1);
        }
...
return 0;
}
```

9.5.3. fclose

Quando abrimos um arquivo devemos fechá-lo. Para tanto devemos usar a função **fclose()**:

```
int fclose (FILE *fp);
```

É importante que se perceba que se deve tomar o maior cuidado para não se "perder" o ponteiro do arquivo. "Perder" neste caso seria se atribuir um valor de um outro ponteiro qualquer ao ponteiro de arquivo (perdendo assim o valor original). É utilizando este ponteiro que vamos poder trabalhar com o arquivo. Se perdermos o ponteiro de um determinado arquivo não poderemos nem fechá-lo. O ponteiro **fp** passado à função **fclose()** determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

9.6. Lendo e Escrevendo Caracteres em Arquivos

9.6.1. putc

Toda vez que estamos trabalhando com arquivos, há uma espécie de posição atual no arquivo. Esta posição, gerenciada pelo compilador, é a posição de onde será lido ou escrito o próximo caracter. Normalmente, num acesso seqüencial a um arquivo, não temos que mexer nesta posição pois quando lemos um caractere a posição no arquivo é automaticamente atualizada. Num acesso randômico teremos que mexer nesta posição (ver fseek()). Protótipo:

```
int putc (int ch,FILE *fp);
```

Escreve um caracter no arquivo.

9.6.2. getc

Retorna um caracter lido do arquivo. Protótipo:

```
int getc (FILE *fp);
```

9.6.3. feof

EOF ("End of file") indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto podemos usar a função **feof()**. Ela retorna não-zero se o arquivo chegou ao EOF, caso contrário retorna zero. Seu protótipo é:

```
int feof (FILE *fp);
```

A seguir é apresentado um programa onde várias operações com arquivos são realizadas. Primeiro o arquivo é aberto para a escrita, e imprime-se algo nele. Em seguida, o arquivo é fechado e novamente aberto para a leitura. Verifique o exemplo.

```
#include <stdio.h
#include <stdlib.h
#include <string.h

void main()
```

```
{
FILE *p;
char c, str[30], frase[80] = "Este e um arquivo chamado: ";
int i;
/* Le um nome para o arquivo a ser aberto: */
printf("\n\n Entre com um nome para o arquivo:\n");
gets(str);

if (!(p = fopen(str,"w")))  /* Caso ocorra algum erro na
abertura do arquivo..*/
  {                         /* o programa aborta
automaticamente */
  printf("Erro! Impossivel abrir o arquivo!\n");
  exit(1);
  }
/* Se nao houve erro, imprime no arquivo, fecha ...*/
strcat(frase, str);
for (i=0; frase[i]; i++)
  putc(frase[i],p);
fclose(p);

/* abre novamente para a leitura e fecha. */
p = fopen(str,"r");
while (!feof(p))
 {
  c = getc(p);
  printf("%c",c);
  }
fclose(p);
}
```

9.7. Outros Comandos de Acesso a Arquivos

9.7.1. ferror

Protótipo:

```
int ferror (FILE *fp);
```

A função **ferror()** se torna muito útil quando queremos verificar se cada acesso a um arquivo teve sucesso. Cada vez que uma função de arquivo é executada, a própria função registra numa variável especial se houve sucesso na operação ou não. Com **ferror()** podemos ter acesso ao conteúdo desta variável: ela retorna não zero se houve algum erro na última função de acesso ao arquivo.

9.7.2. rewind

A função **rewind()** de protótipo

```
void rewind (FILE *fp);
```

retorna a posição corrente do arquivo para o início.

9.7.3. getw

getw() lê no arquivo um inteiro. Seu protótipo é:

```
int getw (FILE *fp);
```

9.7.4. putw

putw() escreve um inteiro num arquivo. Protótipo:

```
int putw (int i,FILE *fp);
```

9.7.5. fgets

Para se ler uma string num arquivo podemos usar **fgets()** cujo protótipo é:

```
char *fgets (char *str,int tamanho,FILE *fp);
```

A string str lida deve ter seu tamanho determinado pelo programador.

9.7.6. fputs

Protótipo:

```
char *fputs (char *str,FILE *fp);
```

Escreve uma string num arquivo.

9.7.7. fread

Podemos escrever e ler blocos de dados. Para tanto temos as funções **fread()** e **fwrite()**. O protótipo de **fread()** é:

```
unsigned fread (void *buffer,int numero_de_bytes,int count,FILE *fp);
```

O buffer é a região de memória na qual serão armazenados os dados lidos. O número de bytes é o tamanho da unidade a ser lida. O contador indica quantas unidades devem ser lidas. Isto significa que o número total de bytes lidos é:

*numero_de_bytes*count*

A função retorna o número de unidades efetivamente lidas. Este número pode ser menor que o valor requisitado quando o fim do arquivo for encontrado.

9.7.8. fwrite

A função **fwrite()** funciona como a sua companheira **fread()**. Seu protótipo é:

```
unsigned fwrite(void *buffer,int numero_de_bytes,int count,FILE *fp);
```

9.7.9. fseek

Para se fazer procuras e acessos randômicos usa-se a função **fseek()**. Esta move o cursor no arquivo de um valor especificado, a partir de um ponto especificado. Seu protótipo é:

```
int fseek (FILE *fp,long numbytes,int origem);
```

O parâmetro de origem determina a partir de onde é que os bytes de movimentação serão contados. Os valores possíveis são definidos por macros no arquivo cabeçalho e são:

Nome	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Ponto corrente no arquivo
SEEK_END	2	Fim do arquivo

Tendo-se definido a partir de onde irá se contar numbytes determina quantos bytes de deslocamento será dado na posição atual.

9.7.10.remove

Protótipo:

```
int remove (char *nome_do_arquivo);
```

Desmancha um arquivo especificado.

O exercício da página anterior poderia ser rescrito usando-se, por exemplo, **fgets**() e **fputs**(), ou **fwrite**() e **fread**(). A seguir apresentamos uma segunda versão que se usa das funções **fgets**() e **fputs**(), e que acrescenta algumas inovações.

```
#include <stdio.h
#include <stdlib.h
#include <string.h


void main()
{
FILE *p;
char str[30], frase[80] = "Este e um arquivo chamado: ",
resposta[80];
```

```
int i;
/* Le um nome para o arquivo a ser aberto: */
printf("\n\n Entre com um nome para o arquivo:\n");
gets(str);

if (!(p = fopen(str,"w")))  /* Caso ocorra algum erro na
abertura do arquivo..*/
  {                         /* o programa aborta
automaticamente */
  printf("Erro! Impossivel abrir o arquivo!\n");
  exit(1);
  }
/* Se nao houve erro, imprime no arquivo, fecha ...*/
fputs(frase, p);
fputs(str,p);
       /* E se o seu disco estiver muito cheio.. */
if (ferror(p)) printf("\nNem tudo sao flores...\n");
fclose(p);

strcat(frase, str);
/* abre novamente para a leitura e fecha. */
p = fopen(str,"r");
fgets(resposta, strlen(frase)+1, p);
printf("\n\n%s\n", resposta);
fclose(p);
remove(str);
}
```

9.8. Fluxos Padrão

Os fluxos padrão em arquivos permitem ao programador ler e escrever em arquivos da maneira padrão com a qual o programador já lia e escrevia na tela.

9.8.1. fprintf

A função **fprintf()** funciona como a função **printf()**. A diferença é que a saída de **fprintf()** é um arquivo e não a tela do computador. Protótipo:

```
int fprintf (FILE *fp,char *str,...);
```

Como já poderíamos esperar, a única diferença do protótipo de **fprintf()** para o de **printf()** é a especificação do arquivo destino através do ponteiro de arquivo.

9.8.2. fscanf

A função **fscanf()** funciona como a função **scanf()**. A diferença é que **fscanf()** lê de um arquivo e não da tela do computador. Protótipo:

```
int fscanf (FILE *fp,char *str,...);
```

Como já poderíamos esperar, a única diferença do protótipo de **fscanf()** para o de **scanf()** é a especificação do arquivo destino através do ponteiro de arquivo.
Talvez a forma mais simples de escrever o programa da página c960.html seja usando **fprintf** () e **fscanf**(). Fica assim:

```
#include <stdio.h
#include <stdlib.h
void main()
{
FILE *p;
char str[80],c;

/* Le um nome para o arquivo a ser aberto: */
printf("\n\n Entre com um nome para o arquivo:\n");
gets(str);
```

```
if (!(p = fopen(str,"w")))  /* Caso ocorra algum erro na
abertura do arquivo..*/
  {                         /* o programa aborta
automaticamente */
  printf("Erro! Impossivel abrir o arquivo!\n");
  exit(1);
  }
/* Se nao houve erro, imprime no arquivo, fecha ...*/
fprintf(p,"Este e um arquivo chamado:\n%s\n", str);
fclose(p);

/* abre novamente para a leitura e fecha. */
p = fopen(str,"r");
while (!feof(p))
 {
  fscanf(p,"%c",&c);
  printf("%c",c);
  }
fclose(p);
}
```

9.8.3. AUTO AVALIAÇÃO

Veja como você está. Escreva um programa que leia uma lista de nomes e idades de um arquivo texto. Prepare um arquivo para ser lido com nomes e idades (use um flag para encerrar). Apresente os dados lidos em forma de tabela na tela. Use as funções de sua preferência (experimente tentar usar mais de uma!).

9.9. Rotinas de Arquivo Compatíveis com o Sistema UNIX

O sistema UNIX é um dos sistemas operacionais mais usados. O C possui uma série de comando de arquivos compatíveis com este S.O.. Vamos ver alguns deles agora. Para maiores informações, não deixe de consultar o manual do seu compilador.
Uma diferença importante entre o sistema já ensinado e o UNIX é que o Unix usa, ao invés do FILE*, um "manipulador de arquivos". Este é do tipo **int**. Mas como o C discutido em nosso curso até agora tem sido o mais ANSI possível (sempre que não isto é documentado), se o seu compilador (ainda que em Unix) é ANSI, tudo dito nas páginas anterior continua valendo. O Unix provê algumas possibilidades a mais. Veja só:

9.9.1. open

Protótipo:

```
int open (char *nome_do_arquivo,int modo,int acesso);
```

Abre um arquivo. Os parâmetros modo e acesso determinam que tipo de uso o programador fará do arquivo. Para uma referência completa sobre os valores válidos para estes parâmetros, consulte o manual do seu compilador.

9.9.2. creat

A função **open()** abre um arquivo já existente. Já a função **creat()** cria um novo arquivo para escrita. Seu protótipo é:

```
int creat (char *nome_do_arquivo,int acesso);
```

9.9.3. close

A função **close()** de protótipo

```
int close (int da);
```

Aqui vale ressaltar que o inteiro passado para a função **close()** é o descritor de arquivo retornado por **open()** ou por **creat()**.

9.9.4. read

Protótipo:

```
int read (int da,void *buffer,unsigned tamanho);
```

Lê um número de bytes especificado em tamanho. Estes bytes são gravados no buffer para serem depois aproveitados pelo programador. da indica o arquivo do qual serão lidos os dados. A função retorna o número de caracteres lidos (este número pode ser diferente do número de caracteres que se pediu que fossem lido caso se tenha, por exemplo, chegado ao fim do arquivo) e -1 em caso de erro.

9.9.5. write

De funcionamento muito parecido com **read()** a função **write()** tem o seguinte protótipo:

```
int write (int da,void *buffer,unsigned tamanho);
```

A função **write()** escreve os caracteres do buffer no arquivo especificado por da. Retorna o número de caracteres que de fato foram escritos. Retorna -1 no caso de erro.

9.9.6. unlink

É o equivalente UNIX da função **remove()** vista na seção 9.7.10. Seu protótipo é:

```
int unlink (char *nome_do_arquivo);
```

O arquivo é deletado e a função retorna zero no caso de sucesso.

9.9.7. lseek

Protótipo:

```
long lseek (int da,long numero_de_bytes,int origem);
```

A função **lseek()** permite a implementação de acesso randômico no UNIX. A função **lseek()** funciona como a função **fseek()** descrita na seção 9.7.9. A origem pode assumir os mesmos valores (SEEK_SET, SEEK_CUR e SEEK_END).

10.AULA 9 - Tipos de Dados Avançados

Já vimos que uma variável é declarada como

tipo_da_variável lista_de_variáveis;

Vimos também que existem modificadores de tipos. Estes modificam o tipo da variável declarada. Destes, já vimos os modificadores **signed**, **unsigned**, **long**, e **short**. Estes modificadores são incluídos na declaração da variável da seguinte maneira:

modificador_de_tipo tipo_da_variável lista_de_variáveis;

como já vimos na seção 3.3.
Vamos discutir agora outros modificadores de tipo.

10.1. Modificadores de Acesso

Estes modificadores, como o próprio nome indica, mudam a maneira com a qual a variável é acessada e modificada.

10.1.1.const

O modificador **const** faz com que a variável não possa ser modificada no programa. Como o nome já sugere é útil para se declarar constantes. Poderíamos ter, por exemplo:

```
const float PI=3.141;
```

Podemos ver pelo exemplo que as variáveis com o modificador **const** podem ser inicializadas. Mas **PI** não poderia ser alterado em qualquer outra parte do programa. Se o programador tentar modificar **PI** o compilador gerará um erro de compilação.
O uso mais importante de **const** não é declarar variáveis constantes no programa. Seu uso mais comum é evitar que um parâmetro de uma função seja alterado pela função. Isto é muito útil no caso de um ponteiro, pois o conteúdo de um ponteiro pode ser alterado por uma função. Para tanto, basta declarar o parâmetro como **const**. Veja o exemplo:

```
#include <stdio.h>
int sqr (const int *num);
main (void)
```

```
{
int a=10;
int b;
b=sqr (&a);
}
int sqr (const int *num)
{
return ((*num)*(*num));
}
```

No exemplo, **num** está protegido contra alterações. Isto quer dizer que, se tentássemos fazer

```
*num=10;
```

dentro da função **sqr()** o compilador daria uma mensagem de erro.

10.1.2.volatile

O modificador **volatile** diz ao compilador que a variável em questão pode ser alterada sem que este seja avisado. Isto evita "bugs" seríssimos. Digamos que, por exemplo, tenhamos uma variável que o BIOS do computador altera de minuto em minuto (um relógio por exemplo). Seria muito bom que declarássemos esta variável como sendo **volatile**.

10.2. Especificadores de Classe de Armazenamento

Estes modificadores de tipo atuam sobre a maneira com a qual o compilador vai armazenar a variável.

10.2.1.auto

O especificador de classe de armazenamento **auto** define variáveis locais. Raramente usado pois todas as variáveis do C são **auto** por definição.

10.2.2.extern

O **extern** define variáveis que serão usadas em um arquivo apesar de terem sido declaradas em outro. Ao contrário dos programas até aqui vistos, podemos ter programas de vários milhares de linhas. Estes podem ser divididos em vários arquivos (módulos) que serão compilados separadamente. Digamos que para um programa grande tenhamos duas variáveis globais: um inteiro **count** e um **float sum**. Estas variáveis são declaradas normalmente em um dos módulos do programa. Por exemplo:

```
int count;
float sum;
main (void)
{
...
return 0;
}
```

Num outro módulo do programa temos uma rotina que deve usar as variáveis globais acima. Digamos que a rotina que queremos se chama **RetornaCount()** e retorna o valor atual de **count**. O problema é que este módulo será compilado em separado e não tomará conhecimento dos outros módulos. O que fazer? Será que funcionaria se fizermos assim:

```
int count;              /* errado */
float sum;
int RetornaCount (void)
{
return count;
}
```

Não. O módulo compilaria sem problema, mas, na hora que fizermos a linkagem (união dos módulos já compilados para gerar o executável) vamos nos deparar com uma mensagem de erro dizendo que as variáveis globais count e sum foram declaradas mais de uma vez. A maneira correta de se escrever o módulo com a função **RetornaCount()** é:

```
extern int count;       /* certo */
extern float sum;
```

```
int RetornaCount (void)
{
return count;
}
```

Assim, o compilador irá saber que **count** e **sum** estão sendo usados no bloco mas que foram declarados em outro.

10.2.3.static

O funcionamento das variáveis declaradas como **static** depende se estas são globais ou locais.
Variáveis globais **static** funcionam como variáveis globais dentro de um módulo, ou seja, são variáveis globais que não são (e nem podem ser) conhecidas em outros módulos. Isto é útil se quisermos isolar pedaços de um programa para evitar mudanças acidentais em variáveis globais. Variáveis locais **static** são variáveis cujo valor é mantido de uma chamada da função para a outra. Veja o exemplo:

```
int count (void)
{
static int num=0;
num++;
return num;
}
```

A função **count()** retorna o número de vezes que ela já foi chamada. Veja que a variável local **int** é inicializada. Esta inicialização só vale para a *primeira* vez que a função é chamada pois **num** deve manter o seu valor de uma chamada para a outra. O que a função faz é incrementar num a cada chamada e retornar o seu valor. A melhor maneira de se entender esta variável local **static** é implementando. Veja por si mesmo, executando seu próprio programa que use este conceito.

10.2.4.register

O computador tem a memória principal e os registradores da CPU. As variáveis (assim como o programa como um todo) são armazenados na memória. O modificador **register** diz ao compilador que a variável em questão deve ser, se possível, usada em um registrador da CPU.
Vamos agora ressaltar vários pontos importantes. Em primeiro lugar, porque usar o **register**? Variáveis nos registradores da CPU vão ser acessadas em um tempo muito menor pois os registradores são muito mais rápidos que a memória. Em segundo lugar, em que tipo de variável usar o **register**? O **register** *não* pode ser usado em variáveis globais. Isto implicaria que um registrador da CPU ficaria o tempo todo ocupado por conta de uma variável. Os tipos de dados onde é mais aconselhado o uso do **register** são os tipos **char** e **int**, mas pode-se usá-lo em qualquer tipo de dado. Em terceiro lugar, o **register** é um pedido que o programador faz ao compilador. Este não precisa ser atendido necessariamente.
Um exemplo do uso do register é dado:

```
main (void)
{
register int count;
for (count=0;count<10;count++)
        {
        ...
        }
return 0;
}
```

O loop **for** acima será executado mais rapidamente do que seria se não usássemos o **register**. Este é o uso mais recomendável para o **register**: uma variável que será usada muitas vezes em seguida.

10.3. Conversão de Tipos

Em atribuições no C temos o seguinte formato:

destino=orígem;

Se o destino e a origem são de tipos diferentes o compilador faz uma conversão entre os tipos. Nem todas as conversões são possíveis. O primeiro ponto a ser ressaltado é que o valor de origem é convertido para o valor de destino antes de ser atribuído e não o contrário.
É importante lembrar que quando convertemos um tipo numérico para outro nós nunca *ganhamos* precisão. Nós podemos perder precisão ou no máximo manter a precisão anterior. Isto pode ser entendido de uma outra forma. Quando convertemos um número não estamos introduzindo no sistema nenhuma informação adicional. Isto implica que nunca vamos *ganhar* precisão.
Abaixo vemos uma tabela de conversões numéricas com *perda* de precisão, para um compilador com palavra de 16 bits:

De	Para	Informação Perdida
unsigned char	char	Valores maiores que 127 são alterados
short int	char	Os 8 bits de mais alta ordem
int	char	Os 8 bits de mais alta ordem
long int	char	Os 24 bits de mais alta ordem
long int	short int	Os 16 bits de mais alta ordem
long int	int	Os 16 bits de mais alta ordem
float	int	Precisão - resultado arredondado
double	float	Precisão - resultado arredondado
long double	double	Precisão - resultado arredondado

10.4. Modificadores de Funções

A forma geral de uma função é, como já foi visto,

tipo_de_retorno nome_da_função (declaração_de_parâmetros)

{

corpo_da_função

}

Uma função pode aceitar um modificador de tipo. Este vai modificar o modo como a função opera na passagem de parâmetros. A forma geral da função ficaria então:

modificador_de_tipo tipo_de_retorno nome_da_função (declaração_de_parâmetros)

{

corpo_da_função

}

O nosso curso não aborda detalhes do funcionamento interno de funções. Para saber mais, consulte o manual do seu compilador ou algum livro especializado.

10.4.1.pascal

Faz com que a função use a convenção de funções da linguagem de programação Pascal. Isto faz com que as funções sejam compatíveis com programas em Pascal.

10.4.2.cdecl

O modificador de tipo **cdecl** faz com que a função use a convenção para funções do C. Raramente é usado pois é o default. Pode-se pensar no **cdecl** como sendo o "inverso" do **pascal**.

10.4.3.interrupt

Diz ao compilador que a função em questão será usada como um manipulador de interrupções. Isto faz com que o compilador preserve os registradores da CPU antes e depois da chamada à função. Mais uma vez este tópico está fora do escopo do curso.

10.5. Ponteiros para Funções

O C permite que acessemos variáveis e funções através de ponteiros! Esta é mais uma característica que mostra a força da linguagem de programação C. Podemos então fazer coisas como, por exemplo, passar uma função como argumento para outra função. Um ponteiro para uma função tem a seguinte declaração:

*tipo_de_retorno (*nome_do_ponteiro)();*

ou

*tipo_de_retorno (*nome_do_ponteiro)(declaração_de_parâmetros);*

Veja que não é obrigatório se declarar os parâmetros da função. Veja um exemplo do uso de ponteiros para funções:

```
#include <stdio.h>
#include <string.h>
void PrintString (char *str,int (*func)());
main (void)
{
char String [20]="Curso de C.";
int (*p)();
p=puts;
PrintString (String,p);
return 0;
}
void PrintString (char *str,int (*func)())
{
(*func)(str);
}
```

No programa acima, a função **PrintString()** usa uma função qualquer **func** para imprimir a string na tela. O programador pode então fornecer não só a string mas também a função que será usada para imprimi-la. No **main()** vemos como podemos atribuir, ao ponteiro para funções **p**, o endereço da função **puts()** do C.

10.6. Alocação Dinâmica

A alocação dinâmica permite ao programador criar variáveis em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado. Esta é outra ferramenta que nos mostra o poder do C. O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**:

No entanto, existem diversas outras funções que são amplamente utilizadas, mas dependentes do ambiente e compilador. Neste curso serão abordadas somente estas funções básicas. Consulte o manual de seu compilador para saber mais sobre as funções disponíveis para alocação dinâmica.

10.6.1.malloc

A função **malloc()** serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

A função toma o número de bytes que queremos alocar (**num**), aloca na memória e retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **malloc()** retorna um ponteiro nulo. Veja um exemplo de alocação dinâmica com malloc():

```
#include <stdio.h>
#include <stdlib.h>

main (void)
{
int *p;
int a;

... /* Determina o valor de a em algum lugar */
```

```
        p=(int *)malloc(a*sizeof(int));
        if (!p)
                {
                printf ("** Erro: Memoria Insuficiente **");
                exit;
                }

        ...

        return 0;
        }
```

No exemplo acima, é alocada memória suficiente para se colocar **a** números inteiros. O operador **sizeof()** retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro **void*** que **malloc()** retorna é convertido para um **int*** pelo cast e é atribuído a **p**. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de **p[0]** a **p[(a-1)]**.

10.6.2.calloc

A função **calloc()** também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
        void *calloc (unsigned int num, unsigned int size);
```

A função aloca uma quantidade de memória igual a **num * size**, isto é, aloca memória suficiente para uma matriz de **num** objetos de tamanho **size**. Retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **calloc()** retorna um ponteiro nulo. Veja um exemplo de alocação dinâmica com calloc():

```
        #include <stdio.h>
        #include <stdlib.h>

        main (void)
        {
        int *p;
        int a;

        ...

        p=(int *)calloc(a, sizeof(int));
        if (!p)
                {
                printf ("** Erro: Memoria Insuficiente **");
                exit;
                }

        ...

        return 0;
        }
```

No exemplo acima, é alocada memória suficiente para se colocar **a** números inteiros. O operador **sizeof()** retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro **void *** que **calloc()** retorna é convertido para um **int *** pelo cast e é atribuído a **p**. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de **p[0]** a **p[(a-1)]**.

10.6.3.realloc

A função **realloc()** serve para realocar memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

A função modifica o tamanho da memória previamente alocada apontada por ***ptr** para aquele especificado por **num**. O valor de **num** pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque **realloc()** pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida. Se **ptr** for nulo, aloca **size** bytes e devolve um ponteiro; se **size** é zero, a memória apontada por **ptr** é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

10.6.4.free

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função **free()** cujo protótipo é:

```
void free (void *p);
```

Basta então passar para **free()** o ponteiro que aponta para o início da memória alocada. Mas você pode se perguntar: como é que o programa vai saber quantos bytes devem ser liberados? Ele sabe pois quando você alocou a memória, ele guardou o número de bytes alocados numa "tabela de alocação" interna. Vamos rescrever o exemplo usado para a função **malloc()** usando o **free()** também agora:

```
#include <stdio.h>
#include <alloc.h>

main (void)
{
int *p;
int a;
...
p=(int *)malloc(a*sizeof(int));
if (!p)
        {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
        }
...

free(p);

...

return 0;

}
```

10.6.5.AUTO AVALIAÇÃO

Veja como você está. Refaça o exemplo desta página, mas ao invés de trabalhar com um vetor de inteiros, use um vetor de strings (ou uma matriz de char, como você preferir). Faça leituras e apresente os resultados na tela.

10.7. Alocação Dinâmica de Vetores e Matrizes

10.7.1.Alocação Dinâmica de Vetores

A alocação dinâmica de vetores utiliza os conceitos aprendidos na aula sobre ponteiros e as funções de alocação dinâmica apresentados. Um exemplo de implementação para vetor real é fornecido a seguir:

```
#include <stdio.h>
#include <stdlib.h>
```

```
float *Alocar_vetor_real (int n)
{
  float *v;        /* ponteiro para o vetor */
  if (n < 1) {  /* verifica parametros recebidos */
     printf ("** Erro: Parametro invalido **\n");
     return (NULL);
     }
  /* aloca o vetor */
  v = (float *) calloc (n+1, sizeof(float));
  if (v == NULL) {
     printf ("** Erro: Memoria Insuficiente **");
     return (NULL);
     }
  return (v);    /* retorna o ponteiro para o vetor */
}

float *Liberar_vetor_real (int n, float *v)
{
  if (v == NULL) return (NULL);
  if (n < 1) { /* verifica parametros recebidos */
     printf ("** Erro: Parametro invalido **\n");
     return (NULL);
     }
  free(v);        /* libera o vetor */
  return (NULL);  /* retorna o ponteiro */
}

void main (void)
{
  float *p;
  int a;
  ...    /* outros comandos, inclusive a inicializacao de a */
  p = Alocar_vetor_real (a);
  ...    /* outros comandos, utilizando p[] normalmente */
  p = Liberar_vetor_real (a, p);
}
```

10.7.2.Alocação Dinâmica de Matrizes

A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, o que é denominado *indireção múltipla*. A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. Um exemplo de implementação para matriz real bidimensional é fornecido a seguir. A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz. Em cada linha existe um vetor alocado dinamicamente, como descrito anteriormente (compondo o segundo índice da matriz).

```
#include <stdio.h>
#include <stdlib.h>

float **Alocar_matriz_real (int m, int n)
{
  float **v;  /* ponteiro para a matriz */
  int   i;    /* variavel auxiliar      */
  if (m < 1 || n < 1) { /* verifica parametros recebidos */
     printf ("** Erro: Parametro invalido **\n");
     return (NULL);
     }
  /* aloca as linhas da matriz */
  v = (float **) calloc (m+1, sizeof(float *));
```

```
  if (v == NULL) {
     printf ("** Erro: Memoria Insuficiente **");
     return (NULL);
     }
  /* aloca as colunas da matriz */
  for ( i = 0; i <= m; i++ ) {
      v[i] = (float*) calloc (n+1, sizeof(float));
      if (v[i] == NULL) {
         printf ("** Erro: Memoria Insuficiente **");
         return (NULL);
         }
      }
  return (v); /* retorna o ponteiro para a matriz */
}

float **Liberar_matriz_real (int m, int n, float **v)
{
  int  i;  /* variavel auxiliar */
  if (v == NULL) return (NULL);
  if (m < 1 || n < 1) {  /* verifica parametros recebidos */
     printf ("** Erro: Parametro invalido **\n");
     return (v);
     }
  for (i=0; i<=m; i++) free (v[i]); /* libera as linhas da
matriz */
  free (v);      /* libera a matriz */
  return (NULL); /* retorna um ponteiro nulo */
}

void main (void)
{
  float **mat;  /* matriz a ser alocada */
  int   l, c;   /* numero de linhas e colunas da matriz */
  ...           /* outros comandos, inclusive inicializacao
para l e c */
  mat = Alocar_matriz_real (l, c);
  ...           /* outros comandos utilizando mat[][]
normalmente */
  mat = Liberar_matriz_real (l, c, mat);
  ...
}
```

10.7.3.AUTO AVALIAÇÃO

Veja como você está. Refaça o programa de multiplicação de matrizes usando alocação dinâmica de memória para as matrizes. Avalie as principais vantagens desta forma de gerenciamento de memória neste caso.

11.AULA 10 - Tipos de Dados Definidos Pelo Usuário

11.1. Estruturas

Uma estrutura agrupa várias variáveis numa só. Funciona como uma ficha pessoal que tenha nome, telefone e endereço. A ficha seria uma estrutura.

11.1.1.Criando

Para se criar uma estrutura usa-se o comando **struct**. Sua forma geral é:

struct nome_do_tipo_da_estrutura

{

tipo_1 nome_1;

tipo_2 nome_2;

...

tipo_n nome_n;

} variáveis_estrutura;

O nome_do_tipo_da_estrutura é o nome para a estrutura. As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo nome_do_tipo_da_estrutura. Vamos criar uma estrutura de endereço:

```
struct tipo_endereco
        {
        char rua [50];
        int numero;
        char bairro [20];
        char cidade [30];
        char sigla_estado [3];
        long int CEP;
        };
```

Vamos agora criar uma estrutura chamada ficha com os dados pessoais de uma pessoa:

```
struct ficha_pessoal
        {
        char nome [50];
        long int telefone;
        struct tipo_endereco endereco;
        };
```

Vemos, pelos exemplos acima, como declarar uma estrutura e que uma estrutura pode fazer parte de outra.

11.1.2.Usando

Vamos agora utilizar as estruturas declaradas na seção anterior para escrever um programa que preencha uma ficha.

```
#include <stdio.h>
#include <string.h>
struct tipo_endereco
        {
        char rua [50];
        int numero;
        char bairro [20];
        char cidade [30];
        char sigla_estado [3];
        long int CEP;
        };
struct ficha_pessoal
        {
        char nome [50];
        long int telefone;
        struct tipo_endereco endereco;
        };
main (void)
{
struct ficha_pessoal ficha;
strcpy (ficha.nome,"Luiz Osvaldo Silva");
ficha.telefone=4921234;
strcpy (ficha.endereco.rua,"Rua das Flores");
ficha.endereco.numero=10;
strcpy (ficha.endereco.bairro,"Cidade Velha");
strcpy (ficha.endereco.cidade,"Belo Horizonte");
```

```
strcpy (ficha.endereco.sigla_estado,"MG");
ficha.endereco.CEP=31340230;
return 0;
}
```

O programa declara uma variável ficha do tipo **ficha_pessoal** e preenche os seus dados. O exemplo mostra como podemos acessar um elemento de uma estrutura: basta usar o ponto (**.**).

11.1.3.Matrizes de estruturas

Um estrutura é como qualquer outro tipo de dado no C. Podemos, portanto, fazer matrizes de estruturas. Vamos ver como ficaria a declaração de uma matriz de 100 fichas pessoais:

```
struct ficha_pessoal fichas [100];
```

Poderíamos então acessar a segunda letra da sigla de estado da décima terceira ficha fazendo:

```
fichas[12].endereco.sigla_estado[1];
```

11.1.4.Atribuindo

Podemos atribuir duas estruturas que sejam do *mesmo* tipo. O C irá, neste caso, copiar uma estrutura na outra. Esta operação não apresenta problemas pois quando declaramos

```
struct ficha_pessoal ficha;
```

ficha não é um ponteiro, mas uma estrutura. Veja as linhas abaixo:

```
void main()
{
struct ficha_pessoal primeira, segunda;
Le_dados(&primeira);
segunda = primeira;
Imprime_dados(segunda);
}
```

São declaradas duas estruturas do tipo *ficha_pessoal*, uma chamada *primeira* e outra chamada *segunda*. Supondo que haja declarada uma função `Le_dados()` que faça a leitura de uma estrutura, admitimos que após a execução da segunda linha de `main()`, a estrutura *primeira* estará preenchida com dados válidos. Os valores de *primeira* são copiados em *segunda* apenas com a expressão de atribuição:

```
segunda = primeira;
```

Todos os campos de primeira serão copiados na ficha chamada segunda. Devemos tomar cuidado com a seguinte declaração:

```
struct ficha_pessoal fichas [100];
```

pois neste caso **fichas** é um ponteiro para a primeira ficha. Se quisermos *a* estrutura completa da n-ésima ficha devemos usar **fichas[n-1]**.

11.1.5.Passando para funções

No exemplo apresentado no item usando, vimos o seguinte comando:

```
strcpy (ficha.nome,"Luiz Osvaldo Silva");
```

Neste comando um elemento de uma estrutura é passado para uma função. Este tipo de operação pode ser feita sem maiores considerações.
Podemos também passar para uma função uma estrutura inteira. Veja a seguinte função:

```
void PreencheFicha (struct ficha_pessoal ficha)
{
...
}
```

Como vemos acima é fácil passar a estrutura como um todo para a função. Devemos observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor. Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função. Mais uma vez podemos contornar este pormenor usando ponteiros e passando para a função um ponteiro para a estrutura.

11.1.6.Ponteiros

Podemos ter um ponteiro para uma estrutura. Vamos ver como poderia ser declarado um ponteiro para as estruturas de ficha que estamos usando nestas seções:

```
struct ficha_pessoal *p;
```

Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C. Há, entretanto, um detalhe a ser considerado. Se apontarmos o ponteiro **p** declarado acima para uma estrutura qualquer e quisermos acessar um elemento da estrutura poderíamos fazer:

```
(*p).nome
```

Este formato raramente é usado. O que é comum de se fazer é acessar o elemento **nome** através do operador seta (**->**). Assim faremos:

```
p->nome
```

A declaração acima é muito mais fácil e concisa. Para acessarmos o elemento **CEP** dentro de **endereco** faríamos:

```
p->endereco.CEP
```

Fácil, não?

11.1.7.AUTO AVALIAÇÃO

Veja como você está. Escreva um programa fazendo o uso de struct's. Você deverá criar um tipo chamado ponto, contendo apenas a posição x e y (inteiros) do ponto. Declare 2 pontos, leia a posição (coordenadas x e y) de cada um e calcule a distância entre eles. Divida o programa através do uso de funções (por exemplo usando uma rotina para a leitura e uma rotina para o cálculo de distância). Apresente no final a distância entre os dois pontos.

11.2. Campos Bit

Campos bit são uma extensão da estrutura simples que permite que acessemos os bits individuais de uma determinada estrutura. Sua forma geral é:

struct nome_do_tipo_da_estrutura

{

tipo_1 nome_1 : tamanho_1;

tipo_2 nome_2 : tamanho_2;

...

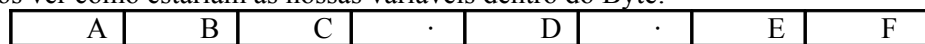
tipo_n nome_n : tamanho_n;

} variáveis_estrutura;

Os tamanhos indicados acima são dados em números de bits. Vamos, por exemplo, declarar uma estrutura de campos bit que nos permite acesso a um byte pedaço por pedaço:

```
struct Byte
        {
        unsigned A : 1;
        unsigned B : 1;
        unsigned C : 2;
        unsigned D : 2;
        unsigned E : 1;
        unsigned F : 1;
        };
```

Vamos ver como estariam as nossas variáveis dentro do Byte:

A	B	C	·	D	·	E	F

Se fizermos então

```
Byte.E
```

estaremos acessando o sétimo bit dentro do byte. No resto, os campos bit funcionam como as estruturas.

11.3. Declaração Union

Uma declaração **union** determina uma *única* localização de memória onde podem estar armazenadas várias variáveis diferentes. A declaração de uma união é semelhante à declaração de uma estrutura:

union nome_do_tipo_da_union

{

tipo_1 nome_1;

tipo_2 nome_2;

...

tipo_n nome_n;

} variáveis_union;

Como exemplo, vamos considerar a seguinte união:

```
union angulo
        {
        float graus;
        float radianos;
        };
```

Nela, temos duas variáveis (**graus** e **radianos**) que, apesar de terem nomes diferentes, ocupam o *mesmo* local da memória. Isto quer dizer que só gastamos o espaço equivalente a um único **float**. Uniões podem ser feitas também com variáveis de diferentes tipos. Neste caso, a memória alocada corresponde ao tamanho da maior variável no union. Veja o exemplo:

```
#include <stdio.h>
#define GRAUS 'G'
#define RAD 'R'
union angulo
{
int graus;
float radianos;
};
void main()
{
union angulo ang;
char op;
printf("\nNumeros em graus ou radianos? ");
scanf("%c",&op);
if (op == GRAUS)
{
ang.graus = 180;
printf("\nAngulo: %d\n",ang.graus);
}
else if (op == RAD)
{
ang.radianos = 3.1415;
printf("\nAngulo: %f\n",ang.radianos);
}
else printf("\nEntrada invalida!!\n");
}
```

Temos que tomar o maior cuidado pois poderíamos fazer:

```
#include <stdio.h>
union numero
        {
        char Ch;
        int I;
        float F;
        };
```

```
main (void)
{
union numero N;
N.graus = 123;
printf ("%f",N.F);
return 0;
}
```

O programa acima é muito perigoso pois você está lendo uma região da memória, que foi "gravada" como um inteiro, como se fosse um ponto flutuante. Tome cuidado! O resultado pode não fazer sentido.

11.4. Enumerações

Numa enumeração podemos dizer ao compilador quais os valores que uma determinada variável pode assumir. Sua forma geral é:

enum nome_do_tipo_da_enumeração {lista_de_valores} lista_de_variáveis;

Vamos considerar o seguinte exemplo:

```
enum dias_da_semana {segunda, terca, quarta, quinta, sexta,
                                        sabado, domingo};
```

O programador diz ao compilador que qualquer variável do tipo **dias_da_semana** só pode ter os valores enumerados. Isto quer dizer que poderíamos fazer o seguinte programa:

```
#include <stdio.h>
enum dias_da_semana {segunda, terca, quarta, quinta, sexta,
                                      sabado, domingo};
main (void)
{
enum dias_da_semana d1,d2;
d1=segunda;
d2=sexta;
if (d1==d2)
        {
        printf ("O dia e o mesmo.");
        }
else
        {
        printf ("São dias diferentes.");
        }
return 0;
}
```

Você deve estar se perguntando como é que a enumeração funciona. Simples. O compilador pega a lista que você fez de valores e associa, a cada um, um número inteiro. Então, ao primeiro da lista, é associado o número zero, o segundo ao número 1 e assim por diante. As variáveis declaradas são então variáveis **int**.

11.5. O Comando sizeof

O operador **sizeof** é usado para se saber o tamanho de variáveis ou de tipos. Ele retorna o tamanho do tipo ou variável em bytes. Mas porque usá-lo se sabemos, por exemplo, que um inteiro ocupa 2 bytes? Devemos usá-lo para garantir portabilidade. O tamanho de um inteiro pode depender do sistema para o qual se está compilando. O **sizeof** é chamado um operador porque ele é substituído pelo tamanho do tipo ou variável *no momento da compilação*. Ele não é uma função. O **sizeof** admite duas formas:

sizeof nome_da_variável
sizeof (nome_do_tipo)

Se quisermos então saber o tamanho de um **float** fazemos **sizeof(float)**. Se declararmos a variável **f** como **float** e quisermos saber o seu tamanho faremos **sizeof f**. O operador **sizeof** também funciona com estruturas, campos bit, uniões e enumerações.
Outra aplicação importante do operador **sizeof** é para se saber o tamanho de tipos definidos pelo usuário. Seria, por exemplo, uma tarefa um tanto complicada a de alocar a memória para um

ponteiro para a estrutura *ficha_pess*oal, criada na primeira página desta aula, se não fosse o uso de **sizeof**. Veja o exemplo:

```
#include <stdio.h>
struct tipo_endereco
{
char rua [50];
int numero;
char bairro [20];
char cidade [30];
char sigla_estado [3];
long int CEP;
};
struct ficha_pessoal
{
char nome [50];
long int telefone;
struct tipo_endereco endereco;
};
void main(void)
{
struct ficha_pessoal *ex;
ex = (struct ficha_pessoal *) malloc(sizeof(struct
      ficha_pessoal));
...
free(ex);
}
```

11.6. O Comando typedef

O comando **typedef** permite ao programador definir um novo nome para um determinado tipo. Sua forma geral é:

typedef antigo_nome novo_nome;

Como exemplo vamos dar o nome de **inteiro** para o tipo **int**:

```
typedef int inteiro;
```

Agora podemos declarar o tipo **inteiro**.

O comando **typedef** também pode ser utilizado para dar nome a tipos complexos, como as estruturas. As estruturas criadas no exemplo da página anterior poderiam ser definidas como tipos através do comando **typedef**. O exemplo ficaria:

```
#include <stdio.h>
typedef struct tipo_endereco
{
char rua [50];
int numero;
char bairro [20];
char cidade [30];
char sigla_estado [3];
long int CEP;
} TEndereco;
typedef struct ficha_pessoal
{
char nome [50];
long int telefone;
TEndereco endereco;
}TFicha;
void main(void)
{
TFicha *ex;
...
}
```

11.7. Considerações finais

Chegamos ao final deste nosso **Curso de Programação de Computadores** utilizando a **Linguagem C**. Esperamos que vocês tenham gostado e aprendido alguma coisa conosco. Agradecemos pela paciência ao acompanhar este longo curso e gostaríamos de contar com a sua colaboração, respondendo à avaliação que irão receber por e-mail (caso estejam matriculados no curso), ou então preenchendo o formulário de avaliação.

O aprendizado de uma linguagem como o C depende de se programar bastante e tentamos fazer isto no nosso curso. Porém, não apresentamos todos os detalhes da linguagem (isto não seria possível em um curso com o tempo que propusemos) e, por isto, sugerimos àqueles que quiserem saber mais que consultem nossa bibliografia buscando lá as referências para o seu estudo.

Boa sorte, bons estudos e divirtam-se com esta fantástica linguagem que agora (esperamos) vocês dominam!

11.8. Bibliografia

Kernighan, B. & Ritchie, D. *C - A linguagem de programação padrão ANSI*. Editora Campus, 1990.
Schildt, H. *Turbo C - guia do usuário* . Editora McGraw-Hill, 1988.
Schildt, H. *C - completo e total* . Editora McGraw-Hill, 1990.
Carpenter, V. *Learn C/C++ today* : http://www.cyberdiem.com/vin/learn.html (uma coleção de referências e tutoriais sobre as linguagens C e C++ disponíveis na Internet)
Mizrahi, V. V. *Treinamento em linguagem C* - Módulos 1 e 2. Editora McGraw-Hill, 1990.
Volta ao sumário.

Dúvidas e Sugestões *gilmar@inf.ufes.br*